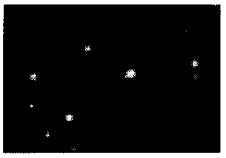
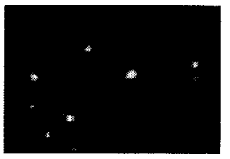


Pleiades Project



Collected Work
1998–1999

Multidisciplinary University Research Initiative

Semantic Consistency in Information Exchange

Office of Naval Research Grant N00014-97-1-0505

Edited by *Iliano Cervesato* and *John C. Mitchell*

Preface

This report collects the papers that were written by the participants of the *Pleiades Project* and their collaborators from September 1998 to August 1999. Its intent is to give the reader an overview of our accomplishments during this central phase of the project. Therefore, rather than including complete publications, we chose to reproduce only the first four pages of each paper. The papers can be integrally retrieved from the World-Wide Web through the provided URLs. A list of the current publications of the *Pleiades Project* is accessible at the URL

`http://theory.stanford.edu/muri/papers.html`.

Future articles will be posted there as they become available.

This report is divided into six parts that reflect the scope of the *Pleiades Project*. These are: Security Protocol Analysis, Real-Time Systems, Result Checking, Complexity, Logic and Programming Languages, and Spatial Control. It contains excerpts from a total of 31 articles.

The *Pleiades Project*, or *Multidisciplinary University Research Initiative* (*MURI*) on *Semantic Consistency in Information Exchange*, is funded by grant number N00014-97-0505 of the Office of Naval Research. Its purpose is to investigate issues of semantic consistency in the transfer of active information, such as executable program components, in information systems that are maintained over time, distributed over many locations, or composed of separate subsystems.

The current participants of the *Pleiades Project* include Iliano Cervesato (Stanford University), *Cynthia Dwork* (IBM Almaden Research Center), *Diana Gordon* (Naval Research Laboratory), *Sampath* Kannan (University of Pennsylvania), *Insup* Lee (University of Pennsylvania), *Patrick Lincoln* (SRI International), *John Mitchell* (Stanford University, principal investigator), *Ronitt Rubinfeld* (Cornell University), *Andre Scedrov* (University of Pennsylvania), *Oleg Sokolsky* (University of Pennsylvania), and *Ulrich Stern* (Stanford University), and graduate students from Cornell University, the University of Pennsylvania, and Stanford University.

Stanford, August 25th 1999
Iliano Cervesato,
John C. Mitchell.

Contents

Part I

Security Protocol Analysis

Patrick Lincoln, John Mitchell, Mark Mitchell, and Andre Scedrov: "Probabilistic polynomial-time equivalence and security protocols", in the proceedings of the World Congress On Formal Methods in the Development of Computing Systems — FM'99, Toulouse, France, September 1999.

Full paper: http: //www.csl.sri.com/~lincoln/papers/fm99.ps

Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov: "Undecidability of bounded security protocols", in the Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP'99 (N. Heintze and E. Clarke, editors), Trento, Italy, July 1999.

Full paper: ftp://www.cis.upenn.edu/pub/papers/scedrov/fmsp99.ps.gz

Iliano Cervesato, Nancy Durgin, Patrick Lincoln, John Mitchell, and *Andre Scedrov:* "A Meta-Notation for Protocol Analysis", in the Proceedings of the Twelfth IEEE Computer Security Foundations Workshop — CSFW'99 (R. Gorrieri editor), pp. 55-69, IEEE Computer Society Press, Mordano, Italy, 2830 June 1999.

Full paper: http: //www.stanford.edu/~iliano/papers/csfw99.ps.gz

Cynthia Dwork and Amit Sahai: 'Concurrent Zero Knowledge: Reducing the Need for Timing Constraints", in the Proceedings of the 18th Annual IACR Crypto Conference - CRYPT0'98, Santa Barbara, CA, August 1999.

Full paper: http://theory.stanford.edu/muri/reports/98-99/cynthia2.ps

Cynthia *Dwork, Moni Naor, and Amit Sahai:* "Concurrent Zero Knowledge", in the Proceedings of the 30th Annual ACM Symposium on the Theory of Computing - STOC'98, Dallas, TX, May 1998.

Full paper: `http://theory.stanford.edu/muri/reports/98-99/cynthia2.ps`

Probabilistic Polynomial-Time Equivalence and Security Analysis

P. Lincoln *,[1], J. Mitchell **,[2], M. Mitchell ***,[2], and A. Scedrov †,[3]

[1] Computer Science Laboratory, SRI International
[2] Department of Computer Science, Stanford University
[3] Department of Mathematics, University of Pennsylvania

Abstract. We use properties of observational equivalence for a probabilistic process calculus to prove an authentication property of a cryptographic protocol. The process calculus is a form of r-calculus, with probabilistic scheduling instead of nondeterminism, over a term language that captures probabilistic polynomial time. The operational semantics of this calculus gives priority to communication over private channels, so that the presence of private communication does not affect the observable probability of visible actions. Our definition of observational equivalence involves asymptotic comparison of uniform process families, only requiring equivalence to within vanishing error probabilities. This definition differs from previous notions of probabilistic process equivalence that require equal probabilities for corresponding actions; asymptotics fit our intended application and make equivalence transitive, thereby justifying the use of the term "equivalence." Our security proof uses a series of lemmas about probabilistic observational equivalence that may well prove useful for establishing correctness of other cryptographic protocols.

1 Introduction

Protocols based on cryptographic primitives are commonly used to protect access to computer systems and to protect transactions over the internet. Two well-known examples are the Kerberos authentication scheme [KNT94, KN93], used to manage encrypted passwords, and the Secure Sockets Layer [FKK96], used by internet browsers and servers to carry out secure internet transactions. Over the past decade or two, a variety of methods have been developed for analyzing and reasoning about such protocols. These approaches include specialized logics such as BAN logic [BAN89], special-purpose tools designed for cryptographic protocol analysis [KMM94], as well as theorem proving [Pau97a, Pau97b] and model-checking methods using general purpose tools [Low96, Mea96, MMS97, Ros95, Sch96].

* Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505.
** Additional support from NSF CCR-9629754.
*** Additional support from Stanford University Fellowship.
† Additional support from NSF Grant CCR-9800785.

In two previous papers [LMMS98, MMS98], we outlined a framework for protocol analysis employing assumptions different from those used in virtually all other formal approaches. Specifically, most formal approaches use a basic model of adversary capabilities which appears to have developed from positions taken by Needham and Schroeder [NS78] and a model presented by Dolev and Yao [DY83]. This set of modeling assumptions treats cryptographic operations as "black-box" primitives, with plaintext and ciphertext treated as atomic data that cannot be decomposed into sequences of bits. Furthermore, as explained in [MMS97, Pau97a, Sch96], there are limited ways for an adversary to learn new information. For example, if a decryption key is sent over the network "in the clear," it can be learned by the adversary. However, it is not possible for the adversary to learn the plaintext of an encrypted message unless the entire decryption key has already been learned. Generally, the adversary is treated as a nondeterministic process that may attempt any possible attack, and a protocol is considered secure if no possible interleaving of actions results in a security breach. The two basic assumptions of this model, perfect cryptography coupled with nondeterministic computation on the part of the adversary, provide an idealized setting in which protocol analysis becomes relatively tractable. However, this model reduces the power of the adversary relative to real-world conditions. As a result, it is possible to prove a protocol correct in this standard model, even when the protocol is vulnerable to simple deterministic attacks.

Our goal is to establish a framework that can be used to analyze protocols (and, potentially, other computer security components) under the standard assumptions of complexity-based cryptography. In [LMMS98], we outlined a refinement of spi-calculus [AG97] that requires a calculus of communicating probabilistic polynomial-time processes and an asymptotic form of observational equivalence. We proposed basic definitions of the key concepts and discussed the potential of this framework by examining some extremely simple protocols. The sequential fragment of our calculus is developed in more detail in [MMS98], where a precise correspondence is proved between a modal-typed lambda calculus and probabilistic polynomial-time computation. In the present paper, we test our basic definitions by considering further applications and develop a more refined probabilistic semantics. Using our improved semantics, we sketch a proof of correctness for a less trivial protocol. Specifically, we prove correctness of a mutual authentication protocol proposed by Bellare and Rogaway [BR94]. This security proof involves some reasoning about a specific form of asymptotic probabilistic observational equivalence for our process calculus. Since, to the best of our knowledge, there has been no previous work on process equivalence up to some error tolerance, this argument and the difficulties we have encountered motivate further investigation into resource-bounded probabilistic semantics and information hiding.

In addition to relying on the basic relation between observational equivalence and security properties developed in the spi-calculus [AG97], we have drawn inspiration from the cryptography-based protocol studies of Bellare and Rogaway [BR94, BR95]. In these studies, a protocol is represented as a set of oracles, each

corresponding to one input-output step by one principal. These oracles are each available to the adversary, which is represented by a probabilistic polynomial-time oracle Turing machine. There are some similarities to our setting, since an adversary has access to each input-output step by a principal by sending and receiving data on the appropriate ports. However, there are some significant technical and methodological differences. In our setting, the protocol and the adversary are both expressed in a formal language. The use of a formal language allows for proof techniques that are based on either the syntactic structure of the protocol or on the semantic properties of all expressible adversaries. We have found the specification method we have adopted from spi-calculus to be relatively natural and more systematic than the specifications used by Bellare and Rogaway. In particular, it appears that our specification of authentication is stronger than the one used in [BR94], requiring us to prove more about the observable properties of a protocol execution. Finally, by structuring our proof around observational equivalence, we are led to develop general methods for reasoning about probabilistic observational equivalence that should prove useful in analyzing other protocols.

2 Process Calculus for Protocol Analysis

A protocol consists of a set of programs that communicate over some medium in order to achieve a certain task. Typically, these programs are parameterized by ***a security parameter*** k, with the idea that increasing the value of k makes the protocol more secure. Often, k is just the length of the keys used in the protocol since it is expected that longer encryption keys make decryption more difficult.

For simplicity, we will consider only those protocols that require some fixed number of communications, independent of the security parameter. In other words, the number of messages sent back and forth before the protocol completes does not increase, even as the security parameter is "cranked up," although the length of the keys used throughout the protocol will increase. This simplification is appropriate for most handshake protocols, key-exchange protocols and authentication protocols. (Many widely-used protocols, including he authentication phase of SSL, serve as examples of "real-world" protocols where the number of messages remains fixed, even as the security parameter is increased.) We are in the process of extending our process calculus to allow looping, which will allow us to deal with more complex protocols, such as those used to prove zero-knowledge. In the present paper, however, we present methods for reasoning about asymptotic observational equivalence that rely on having a fixed bound on the depth of the concurrent process execution tree, and are therefore inappropriate for protocols where the number of messages depends on the security parameter.

Following the work of Abadi and Gordon [AG97], we express security properties of a protocol $\boldsymbol{P}$ by writing an idealized protocol Q which is "patently secure." (Typically, Q requires magic machinery not available in real computational environments, such as perfect random number generators or perfectly

secure communication channels.) Then, we endeavor to show that, for any adversary, the interactions between the adversary and ***P*** have the same observable behavior as the interactions between the adversary and Q. If this condition holds, we can replace the ideal protocol Q with the realizable protocol ***P,*** without compromising security.

The adversary may then be thought of as a ***process context,*** at which point the task of reasoning about security is reduced to the task of reasoning about observational equivalence (also called observational congruence). Our framework is a refinement of the spi-calculus approach in that we replace nondeterministic computation with probabilistic polynomial-time computation while simultaneously shifting from a standard observational equivalence to an asymptotic form of observational equivalence.

2.1 Syntax

The syntax of our probabilistic polynomial-time calculus consists of ***terms*** and ***processes.*** The process portion of the language is a bounded subset of asynchronous π-calculus. However, readers familiar with the traditional n-calculus will note the absence of scope extrusion, or the ability to pass channel names. These omissions are purposeful, and necessary, in order that the expressive power of the calculus correspond to what is commonly believed reasonable in the cryptographic community. It is best to think of the calculus presented here as a notationally familiar means of expressing parallelism and communication, rather than to compare it directly to more traditional forms of π-calculus.

The term portion of the language is used to express all data dependent computation. ,411 terms have natural number type, so the only values communicated from process to process are natural numbers (as is true in the real world). We do not present a formal grammar or semantics for the term calculus (although we did so in [MMS98]). For the purposes of this paper, the important consideration is that the term language be able to express precisely the probabilistic polynomial time functions from integers to integers. (Therefore, an alternative formalism to that employed in [MMS98] would be Turing machine descriptions, together with explicit polynomial time limits, and the understanding that a Turing machine computation that exceeds its time limit outputs zero.) Because the syntax of the term language is unimportant, we use pseudo-code to express terms throughout the paper.

In the grammar below ***P*** varies over processes, ***T*** over terms, x over term variables, and c over a countably infinite set C of channel names. The set of well-formed processes is given by the following grammar:

$$
\begin{array}{lll}
P ::= 0 & & \text{(termination)} \\
\quad (\nu c).P & & \text{(private channel)} \\
\quad c(x).P & & \text{(input)} \\
\quad \bar{c}\langle T \rangle & & \text{(output)} \\
\quad [T = T].P & & \text{(match)} \\
\quad P \mid P & & \text{(parallel composition)}
\end{array}
$$

Undecidability of bounded security protocols*

N.A. Durgin P.D. Lincoln J.C. Mitchell A. Scedrov

Computer Science Lab
SRI International
Menlo Park, CA
lincoln@csl.sri.com

Computer Science Dept.
Stanford University
Stanford, CA 94305-9045
{nad, jcm}@cs.stanford.edu

Mathematics Dept.
University of Pennsylvania
Philadelphia, PA
andre@cis.upenn.edu

(650) 723-8634, Fax:725-4671

Abstract

Using a** multiset **rewriting formalism with existen- tial ***quantification, it is shown that protocol security remains undecidable even when rather*** severe restrictions ***are placed on protocols. In particular, even if data constructors,*** message ***depth, message width, num-*** ber ***of distinct*** roles, ***role length,*** and ***depth*** of ***encryption*** are ***bounded by*** constants, ***secrecy is an undecidable property. If protocols*** are further ***restricted to have no*** new ***data** (nonces), **then*** secrecy ***is*** DEXPTIME-*complete.* ***Both lower bounds are obtained by encoding decision problems from existential Horn theories without func-*** tion ***symbols into our protocol framework. The way that encryption and adversary behavior*** are used in ***the*** reduction ***sheds some light on protocol*** analysis.

1 Introduction

Security protocols are difficult to design and analyze for several reasons. In addition to subtleties involving various cryptographic primitives, most protocols are intended to operate correctly when many instances of the protocol are executed in parallel. A basic client-server protocol, for example, might allow a client to request permission for a session, open that session, halt and reopen, and then finally close the session. This might seem fairly simple, except that several clients may simultaneously request several sessions with the same server, and a malicious attacker may combine data from separate sessions in order to confuse the server. Therefore, verification of a simple protocol may involve analyzing relatively complicated attacks that combine data from any number of valid or aborted runs of the protocol.

Since many published security protocols have subtle flaws, past researchers have devised a variety of formal methods for protocol analysis. Most formal approaches in current use adopt the so-called "Dolev-Yao model" of protocol execution and attack. This model, which involves idealized assumptions about cryptographic primitives and a nondeterministic adversary, appears to have developed from the perspective [13] and a simplified stateless model presented [6].

A multiset rewriting framework for protocol analysis, using the Dolev-Yao model, was introduced [7, 12]. In the present paper, the multiset formalism is used to prove upper and lower complexity bounds on protocol analysis in the Dolev-Yao model. A strong form of undecidability for protocol analysis is presented, along with DEXPTIME completeness for protocols and attacks that do not generate new data once the protocol has begun.

Since most properties of most classes of programs are undecidable, there are many ways to find undecidable classes of protocols. Previous protocol undecidability results [8, 10] have used general forms of protocols that allow agents to take any number of steps and communicate data of unbounded complexity. Since most protocols in use are finite length, we use our rewriting framework to identify a class of finite-length protocols, where each of a finite set of possible roles (such as sender, receiver, server) involves only a finite number of steps. In addition, we bound the use of data constructors, bound the number of symbols that may occur in any message, and bound the depth of encryption by constants. Even in this restrictive case, secrecy

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505, and by NSF Grants CCR-9509931, CCR-9629754, and CCR-9800785 to various authors.

(as well as other properties) is undecidable. The intruder plays a central role by replaying data in a possibly unbounded set of protocol runs. In addition, encryption is used in a crucial way, to limit the power of the intruder to decompose and alter the message parts that are replayed.

2 Protocol Formalism

The protocol formalism we use involves facts and transitions. The facts are first-order atomic formulas, and transitions are given by rewrite rules containing a precondition and postcondition. One property of this formalism is that in applying a rule to a collection of facts, each fact that occurs in the precondition of the rule is removed. This gives us a direct way of representing state transitions, and provides the basis for a connection with linear logic [9]. Another key property is that the postconditions of a rule may contain existentially quantified variables. Following the standard proof rules associated with existential quantification (in natural deduction or sequent-style systems), this provides a mechanism for choosing new values that are distinct from any other in the system.

More formally, our syntax involves terms, facts and rules. If we want to represent a system in this notation, we begin by choosing a vocabulary, or ***first-order*** signature. As usual, the ***terms*** over a signature are the well-formed expressions produced by applying functions to arguments of the correct sort. A *fact* is a first-order atomic formula over the chosen signature. This means that a fact is the result of applying a predicate symbol to terms of the correct sorts. A *state* is a multiset of facts (all over the same signature).

A state transition is a ***rule*** written using two multisets of facts, and existential quantification, in the following syntactic form:

$$F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$$

The meaning of this rule is that if some state S contains facts $F_1, \ldots F_k$, then one possible next state is the state S' that is similar to S, but with:

- facts $F_1, \ldots F_k$ removed,
- $G_1, \ldots G_m$ added, where xi . . x_j are replaced by new symbols.

If there are free variables in the rule, these are treated as universally quantified throughout the rule. In an application of a rule, these variables may be replaced by any terms. A Theory $\mathcal{T}$ is a finite set of facts and rewrite rules of form $1 \rightarrow r$.

The multiset-rewriting notation used in this paper is a first-order Horn fragment of linear logic [9], with existential quantification. Specifically, each transition rule

$$A1, \ldots, A, \longrightarrow \exists \vec{x} . B_1, \ldots, B_m$$

can be written as a linear logic formula

$$A_1 \otimes \ldots \otimes A_n \multimap \exists \vec{x} . B_1 \otimes \ldots \otimes B_m$$

Under this correspondence, every derivation using multiset rewriting corresponds to a linear logic derivation, and conversely.

3 Bounded Protocols

It is relatively straightforward to use the multiset rewriting framework summarized in the preceding section to describe finite-state and infinite-state systems. Using function symbols, it is possible to describe computation over unbounded data types. In particular, it is easy to encode counter machines or Turing machines, implying that secrecy is undecidable. However, the principal authentication and secrecy protocols of interest are all of bounded length, and most use data of bounded complexity (see [3] for a relevant survey).

In order to study finite-length protocols more carefully, we identify the syntactic form of a class owell-founded protocol theories, here called simply *well-**founded** theories.*

3.1 Creation, consumption, persistence

Some preliminary definitions from [2] involve the ways that a fact may be created, preserved, or consumed by a rule. While multiple copies of some facts may be needed in some derivations, we are able to eliminate the need for multiple copies of certain facts.

Definition 1. A rule $1 \rightarrow r$ in a theory $\mathcal{T}$ *creates* ***P*** facts if some $P(\vec{t})$ occurs more times in ***r*** than in 1. A rule $l \rightarrow r$ in a theory $\mathcal{T}$ *preserves* ***P*** facts if every $P(\vec{t})$ occurs the same number of times in r and 1. A rule $1 \rightarrow r$ in a theory $\mathcal{T}$ consumes ***P*** facts if some fact $P(\vec{t})$ occurs more times in l than in r. A predicate ***P*** in a theory $\mathcal{T}$ is *persistent* if every rule in $\mathcal{T}$ which contains ***P*** either creates or preserves ***P*** facts.

As an example, a rule of form

$$P(\vec{x}) \rightarrow P(\vec{y})$$

does not preserve ***P*** facts, since it can be used to create a fact $P(\vec{t})$ and consume a fact $P(\vec{s})$.

Since a persistent fact is never consumed by any rule, there is no need to generate more than one copy of a particular fact – as long as that fact is never needed more than once by a single rule. However, by simple transformation, it is possible to eliminate the need for more than one copy of any persistent fact[2].

Definition 2. A rule $\mathbf{1} \rightarrow \mathbf{r}$ in a theory $\mathcal{T}$ is a ***single-persistent rule*** if all predicates that are persistent in theory $\mathcal{T}$ appear at most once in **1.** A theory $\mathcal{T}$ is a ***uniform theory*** if all rules in $\mathcal{T}$ are single-persistent rules.

Since any theory can be rewritten as a uniform theory, we will assume that all theories discussed from this point forward are uniform theories.

Definition 3. Let **P** be a set of predicates, each persistent in a uniform theory $\mathcal{T}$. Two states S and S' are ***P-similar*** (denoted $S \simeq_P$ S') if, after removing all duplicate persistent ***P*** facts from each state, they are equal multisets.

Lemma 1. ***If*** $S \simeq_P S'$ ***and*** $S \xrightarrow{\mathcal{T}} \mathbf{T}$, ***then*** $\exists T'. T \simeq_P T'$ ***with*** $S' \xrightarrow{\mathcal{T}} \mathbf{T}$**:**

3.2 Protocol theories

In many protocols, there is an implicit or explicit initialization phase that distributes keys or establishes other shared information. Following this initialization phase, each agent may choose to carry out the protocol any number of times, in any combination of roles. For example a principal A may play the role of initiator twice, and responder once, during the course of a single attack. We incorporate these ideas into our formal definitions by letting a protocol theory consist of an initialization theory, a role generation theory, and the disjoint union of bounded subtheories that each characterize a possible role. In order to bound the entire protocol, we must assume that the initialization theory is bounded, and that initialization can be completed prior to the execution of the protocol steps proper. An example of a protocol theory is given in Section 4.

Definition 4. A rule $\mathbf{R} = l \rightarrow \mathbf{r}$ ***enables*** a rule $l' \rightarrow \mathbf{r'}$ if there exist σ, σ' such that some fact $P(\vec{t}) \in$ ***or,*** is also in $\sigma' l'$. A theory $\mathcal{T}$ ***precedes*** a theory $\mathcal{R}$ if no rule in $\mathcal{R}$ enables a rule in $\mathcal{T}$.

In particular, if a theory $\mathcal{T}$ precedes a theory $\mathcal{R}$, then no predicates that appear in the left hand side of rules in $\mathcal{T}$ are created by rules that are in $\mathcal{R}$.

Definition 5. A theory $\mathcal{A}$ is a ***bounded role theory*** if it has an ordered set of predicates, called the ***role states*** and numbered Se, $S_1, \ldots, S_k$ for some k, such that each rule $\mathbf{1} \rightarrow \mathbf{r}$ contains exactly one state predicate $S_i \in \mathbf{1}$ and one state predicate $S_j \in r$, with $i < j$. We call the first role state, Se, an ***initial*** role ***state.***

By defining roles in this way, we ensure that each application of a rule in .A advances the state forward. Each instance of a role can only result in a finite number of steps in the derivation.

If $\mathcal{A}_1, \ldots, \mathcal{A}_k$ is a set of bounded role theories, a role ***generation theory*** is a set of rules of the form

$$P(\vec{s}), Q(\vec{t}), \ldots \rightarrow S_i(\vec{r}), P(\vec{s}), Q(\vec{t}), \ldots$$

where $P(\vec{s}), Q(\vec{t}), \ldots$ is a finite list of persistent facts not involving any role states, and S_i is the initial role state for one of $\mathcal{A}_1, \ldots, \mathcal{A}_k$.

Definition 6. A theory $\mathbf{S} \subset \mathcal{T}$ is a ***bounded sub-theory*** of $\mathcal{T}$ if all facts created by rules **R** in S either contain existentials or are persistent in $\mathcal{T}$.

Definition 7. A theory $\mathcal{P}$ is a ***well-founded protocol theory*** if $\mathcal{P} = \mathcal{I} \uplus \mathcal{R} \uplus \mathcal{A}_1 \uplus \ldots \uplus \mathcal{A}_n$ where 3 is a bounded sub-theory (called the ***initialization*** theory) not involving any role states, $\mathcal{R}$ is a role generation theory involving only facts created by 3 and the initial roles states of $\mathcal{A}_1, \ldots, \mathcal{A}_n$, and $\mathcal{A}_1, \ldots,$ .A, are well-founded role theories, with 3 preceding $\mathcal{R}$ and $\mathcal{R}$ preceding $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

This form allows derivations in a protocol theory to be broken down into three stages – the initialization stage, the role generation stage, and the protocol execution stage.

Lemma 2. ***Given a well-founded protocol theory*** $\mathcal{P} = \mathcal{I} \uplus \mathcal{R} \uplus \mathbf{A}$, ***where*** $\mathcal{I}$ ***is an initialization theory,*** $\mathcal{R}$ ***is a role*** generation ***theory, and*** **A** ***is the disjoint union*** *of one or more* ***bounded*** role ***theories,*** *if* $S \xrightarrow{\mathcal{P}} \mathbf{T}$ ***is a derivation over*** $\mathcal{P}$, ***then there exists a derivation*** $S \xrightarrow{\mathcal{I},\mathcal{R}} S'$ ***and*** $S' \xrightarrow{\mathbf{A}} \mathbf{T}$, ***where all*** rules from ***3 and*** $\mathcal{R}$ are ***applied before any*** rules ***from*** **A.**

3.3 Intruder theory

One of the original motivations for using multiset rewriting for protocol analysis was that this framework allows us to use essentially the same theory for all adversaries for all protocols. In this subsection, we specify the properties of intruder theories that are needed to bound the number of intruder steps needed to produce a given message. As explained in [4], the actions of the standard intruder can be separated into two phases, one in which messages are decomposed into smaller

parts, and one in which these parts are (re)assembled into a message that will be sent to some protocol agent.

In determining the size of a fact, we count the predicate name, each function name, and each variable or constant symbol. For example, fact $P(A, B)$ has size 3, and fact $P(f(A, B), C)$ has size 5.

Definition 8. A rule $R = l \rightarrow r$ is a ***composition rule*** if the size of the largest non-persistent fact in r is greater than the largest non-persistent fact in l. A rule $R = l \rightarrow r$ is a ***decomposition*** rule if the size of the largest non-persistent fact in r is less than the largest non-persistent fact in l.

For example,

$$C(A), C(B) \rightarrow C(\langle A, B \rangle)$$

is a composition rule, and

$$D(\langle A, B \rangle) \rightarrow D(A), D(B)$$

is a decomposition rule.

For the intruder theories we will consider, we allow persistent facts to appear in both the left and right hand sides. So, in general a decomposition rule is of form:

$$D(\langle A, B \rangle), \vec{P}(\ldots) \rightarrow D(A), D(B), \vec{P'}(\ldots)$$

where $\vec{P}$ and $\vec{P'}$ are sets of persistent predicates, with $\vec{P} \subseteq \vec{P'}$ (and similarly for composition rules).

We also need to introduce more complicated decomposition rules, which we call "Decomposition rules with Auxiliary facts". These are pairs of rules of form:

$$D(t), \vec{P}(\ldots) \rightarrow \vec{P'}(\ldots), A(t)$$

and

$$A(t), \vec{Q}(\ldots) \rightarrow \vec{Q'}(\ldots), D(t')$$

where $\vec{P} \subseteq \vec{P'}$, $\vec{Q} \subseteq \vec{Q'}$, and $size(t') <$ ***size(t).*** Here, ***A*** represents an Auxiliary fact (which can appear only in a pair of rules of this form) which is used to amortize the decomposition of $D(t)$ into $D(t')$ across the two rules. Section 4.4 shows an example of this type of decomposition rule, used to allow decrypting an old fact with a newly learned encryption key.

Definition 9. A theory $\mathcal{T}$ is a ***two-phase*** theory if its rules can be divided into three disjoint theories, $\mathcal{T} = \mathcal{S} \uplus \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{S}$ is a bounded sub-theory preceding $\mathcal{C}$ and $\mathcal{D}$, $\mathcal{C}$ contains only composition rules, $\mathcal{D}$ contains only decomposition rules, and no rules in $\mathcal{C}$ precede any rules in $\mathcal{D}$.

Definition 10. A ***normalized derivation*** is a derivation where all rules from the decomposition theory are applied before any rules from the composition theory.

As also shown in [4] in a slightly different context, all derivations in a two-phase theory can be expressed as normalized derivations.

Lemma 3. *If a theory $\mathcal{T}$ is two-phase, and we limit the size of terms, and we limit the* number *of times each existential is instantiated, then there are only finitely* many ***normalized derivations*** in ***the theory.***

In the Dolev-Yao model, the protocol adversary has the capability to overhear, remember, and block messages, to compose/decompose and decrypt/encrypt message fields, and to generate new messages and send them to any other protocol participant. The messages generated by the intruder may be composed of any information supplied to the intruder initially (such as public keys of protocol participants), fresh data generated by the intruder, and data obtained by overhearing or intercepting messages.

The intruder is easily formalized as a set of rewrite rules. While the basic intruder steps remain the same from one protocol to the next, the exact formalization depends on the form of messages used in the protocol. A specific instance of the ***standard intruder*** is described in some detail in Section 4.4.

3.4 Protocol and intruder

Definition 11. Given a well-founded protocol theory $\mathcal{P} = \mathcal{S} \uplus \mathcal{R} \uplus \mathbf{A}$ and a two-phase intruder theory $\mathcal{M}$, a ***standard trace*** is a derivation that has all steps from the $\mathcal{S}$ and $\mathcal{R}$ first, then interleaves steps from the principal theories $\mathbf{A}$ with normalized derivations from the intruder theory $\mathcal{M}$.

Theorem 1. *Let $\mathcal{P}$ be* any ***well-founded protocol*** theory ***and $\mathcal{M}$ be*** any ***two-phase intruder theory.*** *If* ***we bound*** *the* ***number*** *of* ***uses*** *of* ***each existential, and we bound the number*** *of* ***roles generated, and we bound the size*** *of* ***each term, then*** *the* ***set*** *of* ***standard*** traces *of $\mathcal{P} \cup \mathcal{M}$ is finite.*

Later, we show that secrecy is decidable under the conditions of Theorem 1, even without a bound on the number of roles.

4 Example: Needham-Schroeder Public Key Protocol

As an example, we give the full theory of the three-step core of the Needham-Schroeder public-key protocol.

A Meta-notation for Protocol Analysis*

I. Cervesato N.A. Durgin P.D. Lincoln J.C. Mitchell A. Scedrov

Computer Science Lab
SRI International
Menlo Park, CA
lincoln@csl.sri.com

Computer Science Dept.
Stanford University
Stanford, CA 94305-9045
{iliano, nad, jcm}@cs.stanford.edu

Mathematics Dept.
University of Pennsylvania
Philadelphia, PA
andre@cis.upenn.edu

Abstract

Mostformal approaches to security protocol analysis are based on a set of assumptions commonly referred to as the "Dolev-Yao model." In this paper; we use a multiset rewriting formalism, based on linear logic, to state the basic assumptions of this model. A characteristic of our formalism is the way that existential quantification provides a succinct way of choosing new values, such as new keys or nonces. We define a class of theories in this formalism that correspond to finite-length protocols, with a bounded initialization phase but allowing unboundedly many instances of each protocol role (e.g., client, server; initiator, or responder). Undecidability is proved for a restricted class of these protocols, and PSPACE-completeness is claimed for a class further restricted to have no new data (nonces). Since it is a fragment of linear logic, we can use our notation directly as input to linear logic tools, allowing us to do proof search for attacks with relatively little programming effort, and to formally verify protocol transformations and optimizations.

1 Introduction

In the literature on security protocol design and analysis, protocols are commonly described using an informal notation that leaves many properties of a protocol unspecified. For example, a short challenge-response section of a protocol might be written like this:

$$\begin{aligned} A &\longrightarrow B : \{n\}_K \\ B &\longrightarrow A : \{f(n)\}_K \end{aligned}$$

In this notation, a message of the form $\{x\}_y$ consists of a plaintext x encrypted with key y. In this example protocol, Alice chooses a random number ***n*** and sends its encryption to Bob. There is no specific indication of how Bob determines what to send in response, but we can see that Bob returns a message that contains the encryption of f(n). By analogy with familiar protocols, we might assume that he decrypts the message he receives to determine n, then applies f to n and returns the result to Alice (encrypted with the same key).

As written, the protocol description only gives an intended trace or family of traces involving the honest principals. There is no standard way of determining the initial conditions or assumptions about shared information, nor can we see how the principals will respond to messages that differ from those explicitly written. For example, in the case at hand, we must explain in English that K is assumed to be a shared key and that n is generated by Alice. Otherwise, it is a perfectly reasonable interpretation of the two lines above that Alice and Bob initially share a number n. In this case, Alice might send $\{n\}_K$ to Bob, with Bob returning $\{f(n)\}_K$ to Alice only if he receives precisely $\{n\}_K$. While the two readings of the protocol give the same sequence of messages when no one interferes with network transmission, the effects are different if an intruder intercepts the message from Alice to Bob and replaces it with another message. For this reason, the notation commonly found in the literature does not provide a precise basis for security protocol analysis.

Most formal approaches to protocol analysis are based on a relatively abstract set of modeling assumptions, commonly referred to as the "Dolev-Yao model," which appear to have developed from positions taken by Needham and Schroeder [26] and a model presented by Dolev and Yao [11]. In this approach, messages are composed of indivisible abstract values, not sequences of bits, and encryption is modeled in an idealized way. Although the same basic modeling assumptions are used in theorem proving [27], model-checking methods [18, 20, 25, 28, 29] and symbolic search

*Partially supported by DoD MURI 'Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505. and by NSF Grants CCR-9509931, CCR-9629754, and CCR-9800785 to various authors.

tools [17], there does not appear to be any standard presentation of the Dolev-Yao model as it is currently used in a variety of projects. One goal of this paper is to identify the modeling assumptions using the simplest formalism possible, so that the strengths and weaknesses of the Dolev-Yao model can be analyzed, apart from properties of logics or automated tools in which the model is commonly used.

While we began with the idea of creating a new formalism for this purpose, we naturally gravitated toward some form of rewriting, so that protocol execution could be carried out symbolically. In addition to rewriting to effect state transitions, we also needed a way to choose new values, such as nonces or keys. While this seems difficult to achieve directly in standard rewriting formalisms, the proof rules associated with existential quantification appears to be just what is required. Therefore, we have adopted a notation, first presented in [24], that may be regarded as either an extension of multiset rewriting (see, e.g., [3, 4]), with existential quantification, or a Horn fragment of linear logic [14]. A similar fragment of linear logic is used in [16] to represent real-time finite-state systems. Two other efforts using linear logic to model the state-transition aspect of protocols (but not existential quantification for nonces) are [8, 9].

Using this formalism, it is relatively straightforward to characterize the Dolev-Yao intruder and associated cryptographic assumptions. The formalism also seems appropriate for analyzing the complexity of protocol problems, and as a potential intermediate language for systems or approaches that might combine several different protocol analysis tools. We develop a format for presenting finite-length protocols, as the disjoint union of a set of initialization rules and sets of independent transition rules for each protocol participant. Using this form of protocol theory, we show that secrecy is an undecidable property even if data constructors, message depth, message width, number of distinct roles, role length, and depth of encryption are bounded by constants. If any of these restrictions are lifted, prior results, folklore, or a small amount of thought can be used to show undecidability, but we show even for the very small fragment with only nonces secrecy is undecidable. Finally, we have used a linear logic tool, ***LLF*** [5] in two ways. The first is to search executions of a protocol and intruder for protocol flaws. While symbolic search by a logic programming tool is not as efficient as optimized search by tools such as Murφ [10], this method does have the advantage that the input is substantially easier to prepare. The second use of LLF is to formally verify proofs of protocol optimizations. This provides a basis for simplifying search-based analysis and theorem-proving analysis of protocols.

2 Multiset rewriting with existential quantification

2.1 Protocol Notation

The notation we use involves ***facts*** and ***transitions.*** Our facts are first-order atomic formulas, and transitions are given by rewrite rules containing a precondition and postcondition. One important property of this formalism is that in applying a rule to a collection of facts, each fact that occurs in the precondition of the rule is removed. This gives us a direct way of representing state transitions, and provides the basis for the connection with linear logic. Another key property is that the postconditions of a rule may contain existentially quantified variables. Following the standard proof rules associated with existential quantification (in natural deduction or sequent-style systems), this provides a mechanism for choosing new values that are distinct from any other in the system.

More formally, our syntax involves terms, facts and rules. If we want to represent a system in this notation, we begin by choosing a vocabulary, or ***first-order signature.*** This is a standard notion from many-sorted algebra or first-order logic (see, e.g., [13, section 4.3].) As usual, the ***terms*** over a signature are the well-formed expressions produced by applying functions to arguments of the correct sort. A ***fact*** is a first-order atomic formula over the chosen signature. This means that a fact is the result of applying a predicate symbol to terms of the correct sorts. A ***state*** is a multiset of facts (all over the same signature).

A state transition is a ***rule*** written using two multisets of facts, and existential quantification, in the following syntactic form:

$$F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$$

The meaning of this rule is that if some state S contains facts $F_1, \ldots F_k$, then one possible next state is the state S' that is similar to S, but with:

- facts $F_1, \ldots F_k$ removed,
- $G_1, \ldots G_m$ added, where $x_1 \ldots x_j$ are replaced by new symbols.

While existential quantification does not semantically imply there exist "new" values with certain properties, standard proof rules for manipulating existential quantifiers require introduction of fresh symbols (sometimes called Skolem constants), as described below.

If there are free variables in the rule $F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$, these are treated as universally quantified throughout the rule. In an application of a rule,

these variables may be replaced by any terms. To give a quick example, consider the following state, S, and rule, ***R:***

$$\begin{aligned} S &= \{P(f(a)), P(b)\} \\ R &= (\, P(x) \longrightarrow \exists z.\, Q(f(x), z)\,) \end{aligned}$$

One possible next state is obtained by instantiating the rule ***R*** to $P(f(a)) \longrightarrow \exists z.\, Q(f(f(a)),$ z). Applying this rule, we choose a new value, c, for z and replace $P(f(a))$ by $Q(f(f(a)),$ c). This gives us the state

$$S' = \{Q(f(f(a)), c), P(b)\}$$

The importance of existential quantification, for security protocols, is that it provides a direct mechanism for choosing a new value that is different from other values used in the execution of a system. Since many protocols involve choosing fresh nonces, fresh encryption keys, and so on, existential quantification seems like a useful primitive for describing security protocols.

The way that existential quantification is used in our formalism is based on the existential elimination rule from natural deduction. This proof rule is commonly written as follows.

$$[y/x]\phi$$

(3 elim) $\dfrac{\exists x.\phi \qquad \psi}{\psi}$ y not free in any other hypothesis

If we have an existentially quantified axiom $\exists x.\phi$, then this rule says that if we wish to prove some formula ψ, we can choose a new symbol y for the "x that is presumed to exist" and proceed to derive ψ from $[y/x]\phi$. The side condition "y not free in any other hypothesis in the proof of ψ" means that the only hypothesis in the proof of ψ that can contain y is the hypothesis $[y/x]\phi$.

2.2 Simplified Needham-Schroeder

As a means of explaining the Dolev-Yao intruder and encryption models using our notation, we begin with an overly simplified form of the Needham-Schroeder public-key protocol [26]. Without encryption, the core part of the Needham-Schroeder protocol proceeds as follows:

$$\begin{aligned} A &\longrightarrow B : N_a \\ B &\longrightarrow A : N_a, N_b \\ A &\longrightarrow B : N_b \end{aligned}$$

where N_a and N_b are fresh nonces, chosen by Alice ***(A)*** and Bob ***(B),*** respectively.

We can describe this simplified protocol in our notation using the predicates A_i, B_i, N_i for $0 \leq i \leq 3$, with the following intuitive meaning:

$A_i(\ldots)$	Alice in local state i, with the indicated data
$B_i(\ldots)$	Bob in local state i, with indicated data
$N_i(\ldots)$	Network has message i, with indicated data

The data associated with the state of some principal, or a network message, will depend on the particular state or message. Each principal begins in local state 0, with no data. Therefore, predicates A_0 and B_0 are predicates with no arguments. When Alice chooses a nonce, she moves into local state 1. Therefore, predicate A_1 is a predicate of one argument, intended to be the nonce chosen by Alice. Similarly, predicate B_1 has two arguments, the data received from Alice in message one of the protocol and the nonce chosen by Bob for his response.

Using these predicates, we can state the protocol using four transition rules:

$$\begin{aligned} A_0() &\longrightarrow \exists x.A_1(x),\ N_1(x) \\ B_0(), N_1(x) &\longrightarrow \exists y.B_1(x,y), N_2(x,y) \\ A_1(x), N_2(x,y) &\longrightarrow A_2(x,y), N_3(y) \\ B_1(x,y), N_3(y) &\longrightarrow B_2(x,y) \end{aligned}$$

Each rule corresponds to an action by a principal. In the first rule, Alice chooses a nonce, sends it on the network, and remembers the nonce by moving into a local state that retains the nonce value. In the second step, Bob receives a message on the network, chooses his own nonce, transmits it and saves it in his local state. In the third step, Alice receives Bob's message and replies, while in the fourth step Bob receives Alice's final message and changes state.

In Table 1 is a sample trace generated from these rules, beginning from state A_0, B_0. Spacing is used to separate the facts that participate in each step from those that do not.

2.3 Formalizing the intruder

There are two main parts of the Dolev-Yao model as commonly used in protocol analysis. The first is the set of possible intruder actions, applied nondeterministically throughout execution of the protocol. The second is a "black-box" model of encryption and decryption. We explain the intruder actions here, with the encryption model presented in Section 2.4.

The protocol adversary or "intruder" may nondeterministically choose among the following actions at each step:

- Read any message and block further transmission,
- Decompose a message into parts and remember them,
- Generate fresh data as needed,
- Compose a new message from known data and send.

By combining a read with resend, we can easily obtain the effect of passively reading a message without preventing another party from also receiving it.

$B_0()$,	$A_0()$	$\longrightarrow$ $A_1(\mathsf{nA}), N_1(\mathsf{nA})$,	$B_0()$
$A_1(\mathsf{nA})$,	$B_0()$, $N_1(\mathsf{nA})$	$\longrightarrow$ $B_1(\mathsf{nA}, \mathsf{nB})$, $N_2(\mathsf{nA}, \mathsf{nB})$,	$A_1(\mathsf{nA})$
$B_1(\mathsf{nA}, \mathsf{nB})$,	$A_1(\mathsf{nA})$, $N_2(\mathsf{nA}, \mathsf{nB})$	$\longrightarrow$ $A_2(\mathsf{nA}, \mathsf{nB})$, $N_3(\mathsf{nB})$,	$B_1(\mathsf{nA}, \mathsf{nB})$
$A_2(\mathsf{nA}, \mathsf{nB})$,	$B_1(\mathsf{nA}, \mathsf{nB}), N_3(\mathsf{nB})$	$\longrightarrow$ $B_2(\mathsf{nA}, \mathsf{nB})$,	$A_2(\mathsf{nA}, \mathsf{nB})$

Table 1. Sample trace of simplified Needham-Schroeder

In general, the intruder processes data in three phases. The first is to read and decompose data into parts. The second is to remember parts of messages, and the third is to compose a message from parts it remembers. We illustrate the basic form of the intruder actions using one unary network-message predicate N_1 and one binary network-message predicate N_2. Using predicates ***D*** for decomposable messages and ***M*** for the intruder "memory", the basic rules for intercepting, decomposing and remembering messages are

$$\begin{array}{ll} N_1(x) & \longrightarrow D(x) \\ N_2(x,y) & \longrightarrow D(x,y) \\ D(x,y) & \longrightarrow D(x), D(y) \\ D(z) & \longrightarrow \mathrm{M}(\mathrm{z}) \end{array}$$

While the predicate ***D*** may appear to be an unnecessary intermediary here, a protocol with more complicated messages will lead to more interesting ways of destructuring messages. As noted in [7], it is important in proof search to separate the decomposition phase from the composition phase, which is accomplished here using separate ***D*** and C predicates. The rules for composing messages from parts are written using the C, for "composable", predicate as follows:

$$\begin{array}{ll} \boldsymbol{M(x)} & \longrightarrow C(x),\ M(x) \\ C(x) & \longrightarrow N_1(x) \\ C(x),\ C(y) & \longrightarrow C(x,y) \\ C(x,y) & \longrightarrow N_2(x,y) \end{array}$$

The rule for generating new data is

$$\longrightarrow \exists x.M(x)$$

The reason we need the last transition rule (which can be applied any time without any hypothesis) is that the intruder may need to choose new data in order to trick an honest participant in a protocol. This is illustrated in the following attack on the simplified (and obviously insecure) form of the Needham-Schroeder protocol.

For the simplified example at hand, we can compose rules to eliminate the ***D*** and C predicates as follows:

$$\begin{array}{ll} N_1(x) & \longrightarrow M(x) \\ M(x) & \longrightarrow N_1(x),\ M(x) \\ N_2(x,y) & \longrightarrow M(x),\ M(y) \\ M(x), M(y) & \longrightarrow N_2(x,\ y),\ M(x),\ M(y) \\ N_3(x) & \longrightarrow M(x) \\ M(x) & \longrightarrow N_3(x),\ M(x) \end{array}$$

This reduces the number of steps in the trace, shown in Table 2, which has actions of the honest participants in the left column and actions of the intruder indented. For simplicity, duplicate copies of $M(\,)$ facts are not shown, since these have no effect on the execution of the protocol or intruder.

In this attack, the intruder intercepts messages between A and ***B,*** replacing data so that the two principals have a different view of the messages that have been exchanged. Specifically, the intruder replaces Alice's nonce nA by a value ***n*** chosen by the intruder. When Bob responds to the altered message, the intruder intercepts the result and replaces ***n*** by nA so that Alice receives the message she expects.

2.4 Modeling Perfect Encryption

The commonly used "black-box" model of encryption may be written in our multiset notation using the following vocabulary. For concreteness, we discuss public-key encryption. Symmetric or private-key encryption can be characterized similarly. We assume that plaintexts have sort plain and ciphertexts have sort cipher.

- Additional sorts: e-key, d-key
- Predicate: Key-pair(e-key, d-key)
- Function: enc : e-key x pla→ cipher

We could also include a decryption function dec : d-key x cipher→ plain. However, it seems simpler to write protocols using pattern-matching (encryption on the left-hand-side of a rule) to express decryption.

Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints

Cynthia Dwork[1] and Amit Sahai[2]

[1] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120.. E-Mail: daorkQalmaden.ibm.com.

[2] MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA..EMail: amits@theory.lcs.mit.edu***

Abstract. An interactive proof system (or argument) (P, V) is concurrent zero-knowledgeif whenever the prover engages in polynomially many concurrent executions of (P, V), with (possibly distinct) colluding polynomial time bounded verifiers $V_1, \ldots, V_{poly(n)}$, the entire undertaking is zero-knowledge. Dwork, Naor, and Sahai recently showed the existence of a large class of concurrent zero-knowledge arguments, including arguments for all of NP, under a reasonable assumption on the behavior of clocks of nonfaulty processors. In this paper, we continue the study of concurrent zero-knowledge arguments. After observing that, without recourse to timing, the existence of a *trusted* center considerably simplifies the design and proof of many concurrent zero-knowledge arguments (again including arguments for all of NP), we design a preprocessing protocol, making use of timing, to simulate the trusted center for the purposes of achieving concurrent zero-knowledge. Once a particular prover and verifier have executed the preprocessing protocol, any polynomial number of subsequent executions of a rich class of protocols will be concurrent zero-knowledge.

1 Introduction

In order to be useful in the real world, cryptographic primitives and protocols must remain secure even when executed concurrently with other arbitrarily chosen protocols, run by arbitrarily chosen parties, whose identities, goals, or even existence may not be known. Indeed, this setting, characterized in [13] as a ***distributed computing aggregate,*** describes the Internet. Electronic interactions over an aggregate, such as economic transactions, transmission of medical data, data storage, and telecommuting, pose security risks inadequately addressed in computer science research. In particular, the issue of the security of ***concurrent*** executions is often' ignored.

*** Most of this work performed while at the IBM Almaden Research Center. Also supported by a DOD NDSEG doctoral f ellowship, and DARPA grant DABT-96-C-0018.

[1] but not always, e.g. [1] in a different setting

A *zero-knowledge protocol* is supposed to ensure that no information is leaked during its execution. However, when zero knowledge interactions are executed ***concurrently*** both parties can be at risk. Consider the case of zero knowledge proofs: the verifier faces the possibility that the prover with which it is interacting is actually using some concurrently running second interaction as an "oracle" to help answer the verifier's queries – this is the classical chess master's problem. In the case of a proof of knowledge, the interaction may not actually yield a proof. This is an issue of potential ***malleability*** of the interactive proof system, and is addressed in [13]. In contrast, the prover faces the risk that concurrent executions of a protocol with many verifiers may leak information and may not be zero-knowledge in *toto*. In this case the interaction remains a proof but may fail to remain zero knowledge. This issue was first addressed in [16]. To overcome this difficulty, [16] introduce the notion of an (α, β) ***constmint*** for some $\alpha \leq \beta$:

> For any two (possibly the same) non-faulty processors P_1 and P_2, if P_1 measures a elapsed time on its local clock and P_2 measures β elapsed time on its local clock, and P_2 begins its measurement in real time no sooner than P_1 begins, then P_2 will finish after P_1 does.

As [16] points out, an (α, β) constraint is implicit in most reasonable assumptions on the behavior of clocks in a distributed system (e.g., the linear drift assumption). According to the (standard) view that process clocks are under the control of an adversarial scheduler, the (a, β) constraint limits the choices of the adversary to schedules that satisfy the constraints.

Under an (a, β) constraint, [16] shows that there exist constant round concurrent zero-knowledge protocols of various kinds, for example, arguments for any language in NP[2]. In the protocols of [16], processors make explicit use of their local clocks in order to achieve concurrent zero-knowledge. The protocols require that certain timing constraints be met, which limit the kinds of protocol interleavings that can occur.

Our Contribution. In this work, we reduce the need for timing in achieving concurrent zero-knowledge. Specifically, for a rich class of interactive protocols, we are able push all use of timing into a constant round preprocessing phase; furthermore, the real time at which the preprocessing phase between a prover ***P*** and verifier V_1 occurs need not have any relation to the real time when ***P*** and a different verifier V_2 execute the preprocessing. After this preprocessing phase, the prover and the verifier can execute any polynomial number of a rich class of protocols without any further timing constraints, and the whole interaction will be concurrent zero-knowledge. We require the existence of a semantically secure public-key encryption scheme.

By limiting the use of timing to a single initial phase for each (P, V) pair, our methods can reduce the real execution time of protocols. This is because once preprocessing completes the parties never deliberately introduce timing delays in executing steps of future protocols. In contrast, in the protocols of [16] such deliberate delays play a critical role. For many applications, where two parties

[2] under verious standard computational assumptions

will be executing many zero-knowledge protocols, such as authentication with a system, these repeated delays may be expensive. Moreover, as we will see, our approach frequently yields simpler protocols that are easier to prove concurrent zero-knowledge.

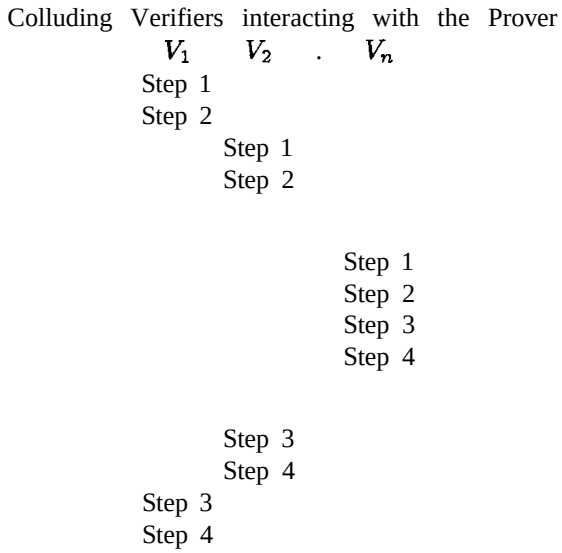


Diagram 1. A troublesome interleaving for concurrent zero-knowledge.

Interleavings of Protocols. The difficulty in achieving concurrent zero-knowledge is due to the existence of certain "bad" interleavings of concurrently executing protocols. The bad interleavings revolve around the difficulty of simulating a transcript of multiple concurrent interactions (recall that the ability to simulate an interaction is the core of the definition of zero-knowledge). Consider the standard (computational) zero-knowledge protocol for 3-colorability[3] [22], which can be based on any information-theoretic commitment scheme.

Generic Zero-Knowledge Argument for 3-Colorability:
1) $V \longrightarrow \boldsymbol{P}$: Information-theoretic commitment to queries.
2) $\boldsymbol{P} \longrightarrow V$: Commitment to graphs and colorings.
3) $V \longrightarrow \boldsymbol{P}$: Open queries.
4) $\boldsymbol{P} \longrightarrow V$: Open queried graphs or colorings, which V then checks are valid.

The standard simulator, having access only to V, produces transcripts of this protocol as follows. First, it receives $\boldsymbol{V}$'s commitment in Step 1. Then, supplying V initially with "garbage" in Step 2, the simulator discovers the queries V committed to through $\boldsymbol{V}$'s Step 3 response. The simulator uses this knowledge to construct graphs and colorings which would fool these particular queries. Then

[3] This is the "parallelized" version that has negligible error while remaining zero-knowledge.

the simulator "rewinds" the interaction to just after Step 1, and supplies V with a commitment to these new graphs and colorings in Step 2. Since V is already committed by Step 1, its Step 3 response cannot change. Thus, the simulator can open the graphs and colorings according to the queries, and V will accept.

This simulator fails in the context of concurrent interactions because of the rewinding. Consider the following interleaving of n colluding verifiers following the generic four-round protocol described above.

An adversary controlling the verifiers can arrange that the Step 1 commitments to queries made by verifiers $V_{i+1}, \ldots, V_n$ can depend on messages sent by the prover in Step 2 of its interaction with V_i. It is a well-known open problem how to simulate transcripts with this interleaving in polynomial time; the difficulty with the straightforward approach is that once the queries in the interaction with V_i are opened (in Step 3), it becomes necessary to re-simulate Step 2 of the interaction with V_i, and therefore the entire simulation of the interaction with verifiers $V_{i+1}, \ldots, V_n$ must be re-simulated. The most deeply nested transaction, with V_n, is simulated roughly 2^n times.

Remark on Commitment Schemes The literature discusses two types of bit or string commitment: ***computational*** and ***znformatzon-theoretic.*** In computational string commitment there is only one possible way of opening the commitment. Such a scheme is designed to be secure against a probabilistic polynomial time receiver and an arbitrarily powerful sender. In information theoretic commitment it is possible to open the commitment in two ways, but the assumed computational boundedness of the sender prevents him from finding a second way. Such a scheme is designed to be secure against an arbitrarily powerful receiver and a probabilistic polynomial time prover. See [13] for a formal definition of computational commitment.

The commitments in Step 1 of the generic zero-knowledge argument must be information-theoretic, meaning that information theoretically nothing is leaked about the committed values. This is for soundness, rather than for zero-knowledge. Our techniques require that the verifier only use computational commitments (for example, as in the 6-round zero-knowledgeargument for NP of Feige and Shamir [19], which we modify for technical reasons).

The Trusted Center Model. Consider a model in which a trusted center gives out signed public key, private key pairs (E, D) of some public key cryptosystem to every user over a secure private channel. As we now explain, in this model arguments such as the one given in [19] can be simulated ***without rewinding,*** provided that the commitments by V are performed using the public key E given to it by the trusted center. This is significant because, if there is no rewinding, then interleavings such as the one described above are not problematic.

The simulator for V simulates its interaction with the trusted center as well as with P. So, the simulator knows the private key D corresponding to the public key E used in V's commitments. Hence, the simulator never has to rewind to learn a committed value. We call such simulations, in which rewinding is

Concurrent Zero-Knowledge

Cynthia Dwork* Moni Naor† Amit Sahai‡

Abstract

Concurrent executions of a zero-knowledge protocol by a single prover (with one or more verifiers) may leak information and may not be zero-knowledge in *toto*; for example, in the case of zero-knowledge interactive proofs or arguments, the interactions remain proofs but may fail to remain zero-knowledge. This paper addresses the problem of achieving concurrent zero-knowledge.

We introduce timing in order to obtain zero-knowledge in concurrent executions. We assume that the adversary is constrained in its control over processors' clocks by what we call an (α, β)-*constraint* for some $a < \beta$: for any two processors P_1 and P_2, if P_1 measures a elapsed time on its local clock and P_2 measures β elapsed time on its local clock, and P_2 starts after P_1 does, then P_2 will finish after P_1 does. We obtain four-round almost concurrent zero-knowledge interactive proofs and perfect concurrent zero-knowledge arguments for every language in *NP*. We also address the more specific problem of Deniable *Authentication*, for which we propose efficient solutions.

1 Introduction

A distributed computing aggregate is a collection of physically separated processors that communicate via a heterogeneous network. To date, research applications of cryptographic techniques to distributed systems have overwhelmingly concentrated on the paradigm in which the system consists of n mutually aware processors trying to cooperatively compute a function of their respective inputs (see e.g. [6, 28]). In distinction with this traditional paradigm, processors in an aggregate do not in general know of all the other members, nor do they generally know the topology of the network. The processors are typically not all trying cooperatively to compute one function or perform a specific set of tasks; in general no coordination is assumed. A prime example of an aggregate is the Internet.

Electronic interactions over an aggregate, such as economic transactions, transmission of medical data, data storage, and telecommuting, pose security risks inadequately addressed in computer science research. In particular, the issue of the security of concurrent executions is often ignored (but not always, e.g., [4] in a different setting). In this paper we address this issue in the context of zero-knowledge interactions (and thus continue research initiated in [16] on zero-knowledge interactions in an aggregate.)

A zero-knowledge protocol is supposed to ensure that no information is leaked during its execution. However, when zero-knowledge interactions are executed *concurrently* both parties can be at risk. Consider the case of zero-knowledge proofs: the verifier faces the possibility that the prover with which it is interacting is actually using some concurrently running second interaction as an "oracle" to help answer the verifier's queries – this is the classic chess master's problem. Thus, for example in the case of a proof of knowledge, the interaction may not actually yield a proof. This is an issue of potential *malleability* of the interactive proof system, and is addressed in [16].

The prover faces the risk that concurrent executions of a protocol by a single prover (with one or more verifiers) may leak information and may not be zero-knowledge in *toto*. (The problem is slightly more general than this; we elaborate in Section 2.) In this case the interaction remains a proof but may fail to remain zero-knowledge. To date, no zero-knowledge proof system has been proven zero-knowledge under concurrent execution. Indeed, recent work of Kilian and Petrank suggests that certain types of four-round interactive proof systems and arguments[1] cannot remain zero-knowledge under concurrent execution [34].

The situation is reminiscent of the case of *parallelizing* the iterations of a zero-knowledge interactive proof *(P, V)*. Such a proof consists of a basic block that is iterated *k* times in order to ensure that the verifier will accept the proof of a false statement with a probability that shrinks exponentially in *k*. For example, if the basic block consists of

*IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120. Research supported by BSF Grant 32-00032-l. E-mail: dwork@almaden.ibm.com.

† Dept. of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel. Some of this work performed while at the IBM Almaden Research Center. Research supported by BSF Grant 32-00032-l. E-mail: naor@wisdom.weizmann.ac.il.

‡MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA. 02139. Most of this work performed while at the IBM Almaden Research Center. Also supported by a DOD NDSEG doctoral fellowship and partially by DARPA grant DABT63-96-C-0018. E-mail: amits@theory.lcs.mit.edu

[1] In an argument the prover is polynomial time bounded.

$P \longrightarrow V$: Commit to M_0, M_1
$V \longrightarrow P : b \in_R \{0,1\}$
$P \longrightarrow V$: open M_b

then the **parallelization** looks like

$P \longrightarrow V$: Commit to $M_0^1, M_1^1, \ldots, M_0^k, M_1^k$
$V \longrightarrow P : b_1 \ldots b_k \in_R \{0,1\}^k$
$P \longrightarrow V$: open $M_{b_1}, \ldots$, Mb,

This natural parallelization is probably not zero-knowledge; indeed Goldreich and Krawczyk have shown that if L is any language having a three-round zero-knowledge (black-box simulatable) proof with negligible probability of error, then $L \in BPP$ (see [26] for this and analogous results for zero-knowledge arguments and constant round Arthur-Merlin games).

In many cases the parallelization can be modified to achieve zero-knowledge by prepending a step in which the verifier commits to $b_1, \ldots, b_k$ *(see* [10, 22, 26]):

1. $V \longrightarrow P$: Commit to $b_1, , b_k$
2. $P \longrightarrow V$: Commit to $M_0^1, M_1^1, , M_0^k, M_1^k$
3. $V \longrightarrow P$: open $b_1, . , b_k$
4. $P \longrightarrow V$: open $M_{b_1}, \ldots, M_{b_k}$

Since the verifier is completely committed to its queries before the prover sends any message, the simulation is easy: commit to random noise, wait until the verifier reveals its queries, then re-wind and choose a new sequence of commitments so that the — now known — queries $b_1, , b_k$ can be answered, and **finish** the simulation. To ensure that the prover's commitments at Step 2 are independent of the verifier's commitments at Step 1, the Step 1 commitments are information-theoretic.

In the concurrent scenario, since there are many verifiers, and since they are not all known to the prover (or provers) before interaction with the first verifier begins, there is no algorithmic way to force all subsequent verifiers to commit to their queries before the **first** interaction begins. (There may be meta-methods, for example, involving certified public keys for the verifiers, but we do not consider such approaches here, and in any case at best such methods move the burden to some initial step.) Consider the following nested interleaving, shown in Diagram 1 below, of n colluding verifiers $V_1, , V_n$ following the generic four-round protocol described above with a single prover.

V_1	V_2	V_n
Step 1		
Step 2		
	Step 1	
	Step 2	
		Step 1
		Step 2
		Step 3
		Step 4
	Step 3	
	Step 4	
Step 3		
Step 4		

Diagram *1.* A troublesome interleaving.

An adversary controlling the verifiers can arrange that the Step 1 commitments to queries made by verifiers $V_{i+1}, ,$ V_n can depend on messages sent by the prover in Step 2 of its interaction with V_i. It is a well-known open problem how to simulate transcripts with this interleaving in polynomial time; the difficulty with the straightforward approach is that once the queries in the interaction with V_i are opened (in Step 3), it becomes necessary to re-simulate Step 2 of the interaction with V_i, and therefore the entire simulation of the interaction with verifiers $V_{i+1}, . , V_n$ must be re-simulated. The most deeply nested transaction, with V_n, is simulated roughly 2" times.

It is possible that the definition of zero-knowledge (i.e., simulateable) is too demanding: interactions that leak no "useful" information may nonetheless not be simulateable. This is the philosophy in [22], and the motivation for *witness indistinguishability.* In the same spirit, we suggest three ideas for exploration:

1. introduce a notion of time to sufficiently restrict the behavior of an adversarial scheduler that the resulting execution is always simulateable (an *explicit* use of time);

2. use "moderately hard" functions to enforce timing constraints *(an implicit* use of time);

3. enhance the power of the simulator or relax the requirements on the output distribution.

Our precise notion of an explicit use of time is an (α, β)-constraint (for some $\alpha \leq \beta$): for any two (possibly the same) non-faulty processors P_1 and P_2, if P_1 measures a elapsed time on its local clock and P_2 measures β elapsed time on its local clock, and if in addition P_2 begins its measurement in real time no sooner than P_1 begins, then P_2 will finish no sooner than P_1 does. An (α, β) constraint is implied by most reasonable assumptions on the behavior of clocks in a system (e.g. the linear drift assumption). The (a, β)-constraint is important for correctness of the protocol for both parties involved, i.e., zero-knowledge in concurrent zero-knowledge proofs and arguments (Protocols II and II) and unforgeability (soundness) and deniability in the Deniable Authentication protocols (Protocols III, IV and V). However, if the time-frame a is too short (not allowing some parties sufficient time to compute and send messages), then the completeness of those protocols is endangered.

It is often possible to partially eliminate the explicit use of time from concurrent zero-knowledge arguments by employing moderately hard functions, possibly with *shortcut* information [17][2]. Intuitively, these functions are used in order to make sure that a certain party operating within in a time limit a may not extract secret information, but an off-line simulator having sufficient time may extract this information; in this sense they supplement the (α, β)-constraint.

In the same spirit, in some protocols we achieve a slightly relaxed notion of zero-knowledge, which we call E-knowledge, requiring that for any polynomial time bounded adversary $\mathcal{A}$ and for any $0 < \varepsilon = o(1)$, there exists a simulator running in time polynomial in the running time of A and ε^{-1}

[2] A short-cut behaves similarly to a trapdoor function, except that the gap between what can be computed having the short-cut and not having the short-cut is polynomial rather than the (conjectured) super-polynomial gap between what can be done having the trapdoor information and not having the trapdoor information.

that outputs simulated transcripts with a distribution ε-indistinguishable from the distribution on transcripts obtained when the prover interacts with (verifiers controlled by) A. (By &-indistinguishable we mean that no polynomial time observer can distinguish the two distributions with advantage better than E.) For a discussion of related topics, see [29].

To understand the significance of ε-knowledge, suppose we have a simulator that runs in time $S(n, 1/\varepsilon, \mathbf{d(n)})$, where d(n) is the running time of an adversary A interacting with the prover on inputs of length n. Suppose further that there is a task whose success can be recognized (e.g., solving an NP search problem or breaking a particular cryptosystem). Suppose that there is a procedure Π that takes part in an &-knowledge proof and then solves the given task. Let T(n) be the total running time of Π (including the interaction and the solving of the task) and let P(n) be the probability that Π succeeds in solving the task. Under the normal definition of zero-knowledge, where the simulator runs in time $S'(n, \mathcal{A}(n))$, we have that without the interaction one can complete the task in time $S'(n, T(n))$, with success probability P(n) $-$ $\mu(n)$, where $\mu(n)$ is negligible.

For our model, without the interaction, one can complete the task in time $S(n, 1/2P(n)$, T(n)), with success probability $P(n) - P(n)/2$ = P(n)/2 (ignoring negligible terms).

Since we assume we can recognize when the task is completed successfully, the claim above follows by a contradiction argument: if the probability were smaller, then this task would be a distinguisher between simulated and real interactions with better than $P(n)/2$ distinguishing power, a contradiction.

This implies in particular that if the original breaking task could be achieved in polynomial-time with an inverse polymomial probability of success after interacting in the ε-knowledge proof, then it will still be achievable in this way without the interaction (although possibly requiring much more time).

Summary of Results: We introduce timing in order to obtain zero-knowledge in concurrent executions. We obtain four-round concurrent E-knowledge interactive proofs and perfect concurrent zero-knowledge arguments for every language in NP under an (α, β) constraint. (Without the timing constraint we are limited by an impossibility result of [34].) The protocol remains zero-knowledge independent of how many different theorems the prover (or provers) is proving in the concurrent interactions (Section 3). In Section 4 we give several examples of protocols for the case of Deniable *Authentication* (discussed next). In particular, Protocols IV and V are concrete and efficient solutions to the problem.

1.1 An Illustrative Example: Deniable Authentication

[16] presents an extremely simple protocol for what is there termed *public key authentication*, a relaxation of digital signatures that permits an authenticator $A\mathcal{P}$ to authenticate messages m to a second party V, but in which the authentication needn't (and perhaps shouldn't!) be verifiable by a third party. To emphasize this last point we use the term *deniable authentication* and strengthen the definition in [16] by insisting on deniability.

Similar to a digital signature scheme, a deniable authentication scheme can convince V that $A\mathcal{P}$ is willing to authenticate m. However, unlike the case with digital signatures, deniable authentication does not permit V to convince a third party that $A\mathcal{P}$ has authenticated m – there is no "paper trail" of the conversation (other than what could be produced by V alone). Thus, deniable authentication is incomparable with digital signatures. Deniable authentication differs from the concepts of undeniable signature introduced in 1989 by Chaum and Van Antwerpen [12] and *chameleon* signature introduced by Krawczyk and Rabin [36], in that deniable authentication is not intended for ultimate adjudication by a third party, but rather to assure V – and only V – of the validity of the message (see [24] for an excellent discussion of work on undeniable signatures). In addition to addressing the privacy needs cited in the literature on undeniable signatures, zero-knowledge public key authentication also provides a solution to a major commercial motivation for undeniable signatures: to provide proof of authenticity of software to authorized/paying customers only. In particular, proving authenticity of the software to a pirate does not in any way help the pirate to prove authenticity of pirate copies of the software to other customers.

Another nice application of deniable authentication is in authenticating "off the record" remarks that are not for attribution. The prover's public key allows a reporter to be certain of the identity of the source while providing plausible deniability.

The notion of non-malleable security for a public key authentication scheme, defined in [16], is analogous to that of *existential unforgeability* under an adaptive chosen *plaintext attack* for signature schemes [32], but we must make sure to take care of *PIM* ("person-in-the-middle") attacks.

In terms of the discussion above, the definition of non-malleable security in this context is concerned with protecting the *verifier* against an imposter trying to impersonate the prover (authenticator) and falsely "authenticating" the message m. A deniable authentication protocol satisfying the definition of non-malleable security clearly also provides some protection for the prover/authenticator; for example, even though the protocol may not be zero-knowledge, it protects any private key used in the authentication to a great extent – otherwise it would be possible to impersonate the authenticator by learning the private authentication key.

The following public key authentication protocol appears in [16] (see also [18]). P's public key is E, chosen according to a non-malleable public key cryptosystem generator. (Roughly speaking, a public key cryptosystem is non-malleable if, for all polynomial time relations R (with certain trivial exceptions), seeing an encryption $E(\alpha)$ "does not help" an attacker to generate an encryption E(P) such that $R(\alpha, \beta)$.) In all our protocols we assume that the message m to be authenticated is a common input, known to both parties. Also, if any message received is of the wrong format, then the protocol is terminated. In particular, if the message received by P in Step 1 below is not an encryption of a string with prefix m, then P terminates the protocol The concatenation of x and y is denoted x o y.

Protocol 0: Public Key Authentication

1. $V \longrightarrow P : \gamma \in_R E(m \circ r), r \in_R \{0,1\}^n$
2. $P \longrightarrow V : r$

Although proved in [16] to be non-malleably secure, Protocol 0 is not zero-knowledge. However, the protocol is easily modified to be zero-knowledge by the addition of a proof of knowledge. (A zero-knowledge interactive proof is a *proof of* knowledge if there is a polynomial time simulator to ***extract*** the information for which knowledge is being proved [21].) Clearly, once it is zero-knowledge the interaction yields no "paper trail" of involvement by the

prover/authenticator, under sequential executions by the same prover/authenticator.

For the modification we use an information-theoretic commitment scheme $K_{g_1,g_2,p}(r)$ [13] to commit to a string r. $K_{g_1,g_2,p}(x)$ is defined as follows: g_1, g_2 generate the same q-sized subgroup of $\mathbb{Z}_p^*$, where q is a large prime dividing p- 1. Given g_1, g_2, and p, commit to x by sending $g_1^x g_2^z$ mod p where $z \in_R \mathbb{Z}_q$. Note that this is distributed uniformly in the subgroup generated by g_1 for all x. Decommitment is by revealing x and z. Note that if the committer knows a such that $g_2 \equiv g_1^\alpha \bmod p$, then the "commitment" can be opened arbitrarily, so the security of the commitment relies on the hardness of finding discrete logarithms modulo p.

Protocol I: SeqZK Deniable Authentication

1. $V \longrightarrow P : E(m \circ r)$, g_1, g_2, p
2. $P \longrightarrow V : K_{g_1,g_2,p}(r)$
3. $V \longrightarrow P : s, r$, where s is the string of random bits used for the encryption in Step 1
4. $P \longrightarrow V$: open commitment

Intuitively, this protocol "should be" zero-knowledge because Step 2 yields no information about r even *information-theoretically*. However, standard simulation techniques fail for concurrent executions; the difficult transcripts are those in which the adversary nests many executions. We do not know if Protocol I remains zero-knowledge under concurrent executions by the same AP. Similarly, if there is a collection of APs, all using the same secret key, concurrent executions of the protocol may not be zero-knowledge.

Related work. For a discussion of attempts to construct parallel zero-knowledge protocols, see [3] and Goldreich, Chapter 6 [25]. The problem of concurrent zero-knowledge was considered by several groups including Kilian and Petrank [34] and Bellare, Impagliazzo and Jakobsson [2]. The use of timing considerations to ensure soundness and zero-knowledge is new. The only work we aware of that uses timing in zero-knowledge protocols is Brands and Chaum [11], in which very accurate timing is needed in order to prevent person-in-the-middle attacks by distant processors. Note that timing has been suggested as a cryptanalytic tool – the best example is Kocher's timing attack [35] – so it follows that any implementation of a cryptographic protocol must be time aware in some sense. The use of moderately hard functions was introduced by Dwork and Naor [17]. The application as time-capsule was considered by Bellare and Goldwasser [1]. The notion of "time-lock puzzles" is discussed by Rivest, Shamir, and Wagner [39].

2 Model and Definitions

Timing. In our protocols, processors (or machines) use local clocks to measure elapsed time. In any execution the adversary has control over the timing of events, subject to an (a, β)-constraint (for some $\alpha \leq \beta$): for any two (possibly the same) non-faulty processors P_1 and P_2, if P_1 measures α elapsed time on its local clock and P_2 measures β elapsed time on its local clock, and if in addition P_2 begins its measurement in real time no sooner than P_1 begins, then P_2 will finish no sooner than P_1 does. The particular constraint may vary between protocols and must be stated explicitly as part of the description of any protocol.

Zero-Knowledge and Concurrent Zero-Knowledge In the original "black box" forumulation of. zero-knowledge proof systems [31], an interactive proof system (P,V) for a language L is computational (or perfect) *zero-knowledge* if there exists a probabilistic, expected polynomial time oracle machine S, called the simulator, such that for every probabilistic polynomial time verifier strategy V^*, the distributions $(P, V^*)(x)$ and S^{V^*} (x) are computationally indistinguishable (or identical) whenever x $\in L$. Here, formally, the machine V'is assumed to take as input a partial conversation transcript, along with a random tape, and output the verifier's next response. This defintion also holds in the case of arguments or computationally-sound proofs, where the prover and verifier are both probabilistic polynomial time machines.

To investigate preservation of zero-knowledge in a distributed setting, we consider a probabilistic polynomial time adversary that controls many verifiers simultaneously. Here, the adversary A will take as input a partial conversation transcript of a prover interacting with several verifiers concurrently. Hence the transcript includes with each message sent or received by the prover, the local time on the prover's clock at which the event occurred. The output of A will either be a tuple (receive, V, a, t), indicating that P receives message a from V at time t on P's local clock, or (send, V, t), indicating that P must send a message to V at time t on P's local clock. The adversary must output a local time for P that is greater than all the times given in the transcript that was input to A (the adversary cannot rewind P), and standard well-formedness conditions must apply. If these conditions are not met, this corresponds to a non-real situation, so such transcripts are simply discarded. Note that we assume that if the adversary specifies a response time t for the prover that violates a timing constraint of the protocol with V, the prover should answer with a special null response which invalidates the remainder of the conversation with verifier V. The distribution of transcripts generated by an adversary A interacting with a prover P on common input x is denoted $(P \leftrightarrow \mathrm{d})(x)$.

An argument or proof system (P, V) for a language L is computational (or perfect) *concurrent zero-knowledge* if there exists a probabilistic, expected polynomial time oracle machine S such that for every probabilistic polynomial time adversary **A,** the distributions $(P \leftrightarrow \mathrm{d})(x)$ and $S^{\mathcal{A}}(x)$ are computationally indistinguishable (or identical) whenever $x \in L$.

An argument or proof system (P, V) for a language L is computational *concurrent E-knowledge* if for every $\varepsilon > 0$, there exists an oracle machine S, such that for every probabilistic polynomial time adversary A, the running time of $S^{\mathcal{A}}$ is polynomial in n and $1/\varepsilon$, and any probabilistic polynomial time machine B that attempts to distinguish between the distributions $(P \leftrightarrow \mathrm{d})(x)$ and $S^{\mathcal{A}}(x)$ will have advantage at most ε whenever $x \in L$.

When the prover acts honestly and follows the protocol, it does not matter if there is a single entity that is acting as the prover for all verifiers, or if there are many entities that are acting as provers for subsets of the verifiers, since the actions of the provers would be the same, and in our model, the timing of events is controlled by the adversary.

NIZK. In anon-interactive zero-knowledge (NIZK) proof [5, 8, 9, 23] the prover P and verifier V have a common input x and also share a random string σ, called the reference string, of length polynomial in the length of x. To convince the verifier of the membership of x in some fixed NP language L,

Part II

Real-Time Systems

Hanêne Ben-Abdallah, Insup Lee, and Oleg *Sokolsky*: "Specification and Analysis of Real-Time Systems with PARAGON", to appear in the Annals of Software Engineering, Baltzer Science Publishers, 1999.

Full paper: `http://www.cis.upenn.edu/~sokolsky/ase99.ps.gz`

Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan: "Runtime Assurance Based On Formal Specifications", in the proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications — PDPTA'99 (H. Arabnia et al., editors), CSREA Press, Las Vegas, NV, 28 June – 1 July 1999.

Full paper: `http://www.cis.upenn.edu/~rtg/mac/doc/99pdpta.ps`

Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky: "Formally Specified Monitoring of Temporal Properties", in the Proceedings of the European Conference on Real-Time Systems — ECRTS'99, pp. 114–121, York, UK, 9911 June 1999.

Full paper: `http: //www.cis.upenn.edu/~rtg/mac/doc/99ecrts.ps`

Moonjoo Kim, Mahesh Viswanathan, *Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky:* "MaC: A Framework for Run-time Correctness Assurance of Real-Time System", Technical Report MS-CIS-98-37, Department of Computer and Information Sciences, University of Pennsylvania, December 1998

Full paper: `http: //www.cis.upenn.edu/~rtg/mac/doc/mac.ps`

Hanêne Ben-Abdallah and Insup Lee: "A Graphical Language with Formal Semantics for the Specification and Analysis of Real-Time Systems", Special Issue of Integrated Computer-Aided Engineering on Real-time Engineering Systems, Vol. 5, No. 4, IOS Press, 1998.

Full paper:
ftp://ftp.cis.upenn.edu/pub/rtg/Paper/Full_Postscript/icae98.ps.gz

Hee-Hwan Kwak, Insup Lee, and Oleg Sokolsky: "Parametric Approach to the Specification and Analysis of Real-time System Designs based on ACSR-VP", in the proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems, Carmel, CA, 27–29 October 1998.

Full paper:
ftp://ftp.cis.upenn.edu/pub/rtg/Paper/Full_Postscript/monterrey98.ps.gz

Hee-Hwang Kwak, *Jin-Young Choi, Insup Lee, and Anna Philippou:* "Symbolic Weak Bisimulation for Value-Passing Calculi", Technical report MS-CIS-98-22, Department of Computer and Information Science, University of Pennsylvania, May 1998.

Full paper:
http://www.cis.upenn.edu/~techreports/reports/MS-CIS-98-22.ps

Specification and Analysis of Real-Time Systems with PARAGON

Oleg Sokolsky, Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, U.S.A.
sokolsky@saul.cis.upenn.edu

Hanêne Ben-Abdallah
Département d'Informatique
FSEG
Université de Sfax
B.P. 1088
3018 Sfax, Tunisia

Abstract

This paper describes a methodology for the specification and analysis of distributed real-time systems using the toolset called PARAGON. PARAGON is based on the Communicating ***Shared Resources*** paradigm, which allows a real-time system to be modeled as a set of communicating processes that compete for shared resources. PARAGON supports both visual and textual languages for describing real-time systems. It offers automatic analysis based on state space exploration as well as user-directed simulation. Our experience with using PARAGON in several case studies resulted in a methodology that includes design patterns and abstraction heuristics, as well as an overall process. This paper briefly overviews the communicating shared resource paradigm and its toolset PARAGON, including the textual and visual specification languages. The paper then describes our methodology with special emphasis on heuristics that can be used in PARAGON to reduce the state space. To illustrate the methodology, we use examples from a real-life system case study.

1 INTRODUCTION

As software systems become more complex and safety-critical, it is vitally important to ensure reliability properties of these systems. Most complex safety-critical systems are distributed and must function in real-time. ***Formal methods*** have been proposed to aid in development of safety-critical systems. They allow users to specify systems precisely and reason about them in mathematical terms. A variety of methods for dealing with hardware and software systems aimed at distributed and real-time systems have been developed. They include state machines, Petri nets, logics, temporal logics, process algebras and timed automata; the summary of existing approaches and directions for future research can be found in [Clarke and Wing 1996; Cleaveland and Smolka 1996]. As formal methods become more mature and their benefits for development of large systems can be clearly demonstrated, they are being increasingly accepted by the industry.

Most industrial designs yield specifications with very large state spaces. Therefore, tools for mechanical analysis of large specifications are essential for successful application of formal methods in industry. A number of tools based on formal methods have been put forward in the last several years in an effort to increase the usability of formal methods especially within the industrial community. Among the tools that are most widely available are the Concurrency Workbench [Cleaveland *et* al. 1993], Spin [Holzmann 1991], SMV [McMillan 1993]. Analysis of real-time systems is supported by COSPAN [Hardin ***et al.*** 1996], Kronos [Daws ***et*** al. 1995], and Uppaal [Bengtsson ***et al.*** 1995], to name just a few.

Even with tool support, most specifications of real-life systems are too large to be analyzed by brute force. Analysis of large systems is impossible without abstractions and simplifications that serve to reduce an infinite, or finite but unmanageable, state space of the system's specification. Users of each formalism and supporting tools employ a number of abstraction heuristics that help in creating manageable specifications of large-scale systems. Some of the used heuristics are specific to the formalism or the tool, while others are applicable to several related methods. Often when case studies are described, such heuristics are left out or mentioned only briefly. We think it is worth while to make these heuristics explicit for the benefit of future users of formal method tools.

This paper describes a methodology for the specification and analysis of distributed real-time systems using a toolset PARAGON. We describe the process of constructing a formal specification from an informal description of the system, and some of the specification patterns often observed in this process. In addition, we summarize heuristics commonly employed by PARAGON that are aimed at reducing the state space of specifications.

PARAGON is based on the process algebra ACSR [Lee ***et al.*** 1994] and related formalisms. Process algebras, such as CCS [Milner 1989], CSP [Hoare 1985] and ACP [Bergstra and Klop 1985], have been devel-

oped to describe and analyze communicating, concurrently executing systems. A process algebra consists of a concise language, a precisely defined operational semantics, and a notion of equivalence. The language is based on a small set of operators and a few syntactic rules for constructing a complex process from simpler components. The operational semantics describes the possible execution steps that a process can take, i.e., a process specification can be executed, and serves as the basis for various analysis algorithms.

The notion of equivalence captures when two processes behave identically, i.e., they have the same execution steps. To verify a system using a process algebra, one writes a requirements specification as an ***abstract*** process and a design specification as a ***detailed*** process. The correctness can then be established by showing that the two processes are equivalent. The most salient aspect of process algebras is that they support the ***modular*** specification and verification of a system. This is due to the algebraic laws that form a compositional proof system, and thus it is possible to verify the whole system by reasoning about its parts. Process algebras without the notion of time are now used widely in specifying and verifying concurrent systems.

To expand the usefulness to real-time systems, several real-time process algebras have been developed by adding the notion of time and including a set of timing operators to process algebras. In particular, these real-time process algebras provide constructs to express delays and timeouts, which are two essential concepts to specify temporal constraints in real-time systems.

Algebra of Communicating Shared Resource (ACSR) introduced by [Lee ***et*** al. 1994], is a timed process algebra which can be regarded as an extension of CCS. It enriches the set of operators, introducing constructs to capture common real-time design notions such as resource sharing, exception and interrupt handling. ACSR supports the notions of resources, priorities, interrupt, timeout, and process structure. The notion of real time in ACSR is quantitative and discrete, and is accommodated using the concept of timed actions. The execution of a timed action takes one time unit and consumes a set of resources defined in the timed action during that one time unit period. The execution of a timed action is subject to the availability of resources it uses. The contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously.

ACSR is an extension of another real-time process algebra CCSR [Gerber and Lee1994], which shares many aspects of ACSR. In particular, CCSR was the first process algebra to support the notions of both resources and priorities. CCSR, however, lacks instantaneous synchronization since all actions take exactly one time unit. ACSR extends CCSR with the notion of instantaneous events and synchronization, and includes a set of laws complete for finite state processes[Brémond-Grégoire ***et al.*** 1997]. To promote the use of ACSR in the specification and analysis of real-time systems, we have implemented a tool VERSA [Clarke ***et*** al. 1995]. PARAGON is a toolset that extends the capability of VERSA by a providing graphical user

Runtime Assurance Based On Formal Specifications*

I. Lee†, S. Kannan, M. Kim, 0. Sokolsky, and M. Viswanathan
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 23, 1999

Abstract

We describe the Monitoring and Checking (MaC) framework which assures the correctness of the current execution at run-time. Monitoring is performed based on a formal specification of system requirements. MaC bridges the gap between formal specification and verification, which ensures the correctness of a design rather than an implementation, and testing, which partially validates an implementation. An important aspect of the framework is a clear separation between implementation-dependent description of monitored objects and high-level requirements specification. Another salient feature is automatic instrumentation of executable code. The paper presents an overview of the framework and two languages to specify monitoring scripts and requirements, and briefly explain our on-going prototype implementation.

1 Introduction

Much research in the past two decades concentrated on methods for analysis and validation of distributed and real-time systems. Important results have been achieved, in particular, in the area of formal verification [4]. Formal methods of system analysis allow developers to specify their systems using mathematical formalisms and prove properties of these specifications. These formal proofs increase confidence in correctness of the system's behavior. Complete formal verification, however, has not yet become a prevalent method of analysis. The reasons for this are twofold. First, the complete verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, verification results apply not to system implementations, but to formal models of these systems. That is, even if a design has been formally verified, it still does not ensure the correctness of a particular implementation of the design. This is because an implementation often is much more detailed, and also may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of the design that has been verified. One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a pre-determined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation on all possible input sequences. Consequently, when a system is running, it

*This research was supported in part by NSF CCR-9619910, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, and ONR N00014-97-1-0505 (MURI)

†Corresponding Author. Insup Lee, email: lee@cis.upenn.edu; fax: +1(215) 573-3573

is hard to guarantee whether or not the current execution of the system is correct using the two traditional methods. Therefore, the approach of continuously monitoring a running system has can be used to fill the gap between these two approaches.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly with respect to a formal requirements specification. The use of formal methods is the salient aspect of our approach. We concentrate on the following two issues: (1) how to map high-level abstract events that are used in requirement specification to low-level activities of a running system, and (2) how to instrument the code to extract and detect necessary low-level activities. We assume that both requirement specifications and the system implementation are available to us.

The major phases of the framework are as follows: (1) system requirements are formalized; at the same time, a monitoring script is constructed, which is used to instrument the code and establish a mapping from low-level information into high-level events; (2) at run-time, events generated by the instrumented system are monitored for compliance with the requirements specification. The run-time monitoring and checking (MaC) architecture consists of three components: filter, event recognizer, and run-time checker. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented code. The filter sends this information to the event recognizer, which converts it into high-level events and conditions and passes them to the run-time checker.

Each event delivered to the checker has a timestamp, which reflects the actual time of the occurrence of the event. This enables us to monitor real-time properties of the system. Timestamps are assigned to events by the event recognizer based on the clock readings provided by the filter. The run-time checker checks the correctness of the system execution thus *far* according to a requirements specification of the system, based on the information it receives from the event recognizer, and on the past history. The checker can combine monitoring of behavioral correctness of the system control flow with program checking [2] for numerical computations. This integrated approach is a unique feature of the proposed framework. The current prototype implementation of the MaC framework supports the monitoring of a system written in Java. Instrumentation is performed automatically, directly in Java bytecode.

Related work. Computer systems are often monitored for performance measurement, evaluation and enhancement as well as to help debugging and testing [17]. Lately, there has been increasing attention from the research community to the problem of designing monitors that can be used to assure the correctness of a system at runtime. The "behavioral abstraction" approach to monitoring was pioneered by Bates and Wileden [1]. Although their approach lacked formal foundation, it provided an impetus for future developments. Several other approaches pursue goals that are similar to ours. The work of [5] addresses monitoring of a distributed bus-based system, based on a Petri Net specification. Since only the bus activity is monitored, there is no need for instrumentation of the system. The authors of [16] also consider only input/output behavior of the system. In our opinion, instrumentation of key points in the system allows us to detect violations faster and more reliably, without sacrificing too much performance. The test automation approach of [14] is also targeted towards monitoring of black-box systems without resorting to instrumentation. Additionally, we aim at using the MaC framework beyond testing, during real system executions. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [15]. The user manually annotates an Ada program with constructs from ANNA, a formal specification language. Mok and Liu [12] proposed an approach

for monitoring the violation of timing constraints written in the specification language based on Real-time Logic as early as possible with low-overhead. The framework proposed in this paper does not limit itself to any particular kind of monitored properties. In [10], an elaborate language for specification of monitored events based on relational algebra is proposed. Similarly to our approach, the authors try to minimize effects of instrumentation on run-time performance, and to reduce the instrumentation cost through automated instrumentation. Their goal, however, goes beyond run-time monitoring. For our purposes, a simpler and easier to interpret event description language of MaC appears to be more appropriate.

The paper is organized as follows. Section 2 presents an overview of the framework. Section 3 informally presents the language for monitoring scripts and requirements specifications. Section 4 briefly overviews a prototype implementation of the MaC framework as well as the current future plans. More complete and formal treatment of the MaC framework is given in [9].

2 Overview of the MaC Framework

The MaC framework aims at run-time assurance monitoring of real-time systems. The structure of the framework is shown in Figure 1. The framework includes two main phases: (1) before the system is run, its implementation and requirement specification are used to generate run-time monitoring components; (2) during system execution, information about the running system is collected and matched against the requirements.

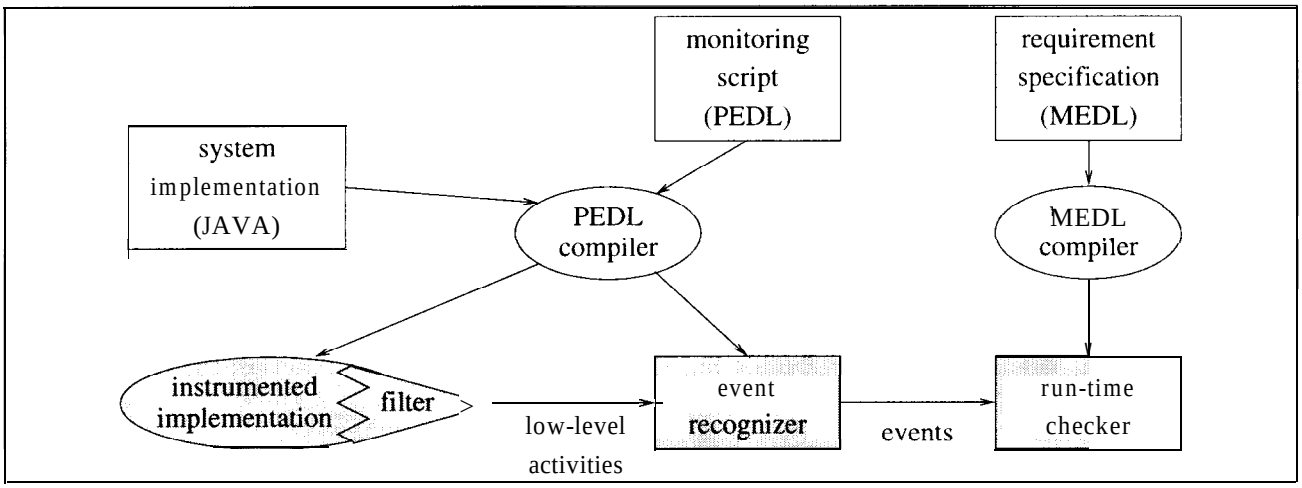


Figure 1: Overview of the MaC framework

A major task during the first phase (indicated by clear boxes in Figure 1) is to provide a mapping between high-level events used in the requirement specification and low-level state information extracted during execution. They are related explicitly by means of a *monitoring script*. The monitoring script describes how events at the requirements level are defined in terms of monitored states of an implementation. For example, in a gate controller of a railroad crossing system, the requirements may be expressed in terms of the event train-in-crossing. The implementation, on the other hand, stores the train's position with respect to the crossing in a variable train-position. The monitoring script in this case can define the event as condition train-position $<$ 800. The language of monitoring script (described in Section 3) has limited expressive power in order to ensure fast recognition of events.

The monitoring script is used to generate a jilter and an *event* recognizer automatically. The filter instruments the implementation to extract the necessary state information at run-time. The event recognizer receives state information from the filter and determines the occurrences of event according to the event definition in the script. Also, a *run-time checker* is generated from the formal requirements. The requirement specification uses events defined in the monitoring script.

During the run-time phase (shaded boxes in Figure 1), the instrumented implementation is executed while being monitored and checked against the requirements specification. The filter sends relevant state information to the event recognizer, which determines the occurrence of events. These events are then relayed to the run-time checker to check adherence to the requirements.

Filter. A filter is a set of program fragments that are inserted into the implementation to instrument the system. The essential functionality of a filter is to keep track of changes to monitored objects and send pertinent state information to the event recognizer. Instrumentation is performed directly on the executable code (bytecode, in the case of Java). Instrumentation is automatic, which is made possible by the low-level description in the monitoring script.

Event Recognizer. The event recognizer is the part of the monitor that detects an event from values of monitored variables received from the filter. Events are cognized according to a monitoring script (written in PEDL) and recognized events are sent to the run-time checker. Each event is supplied with a timestamp that can be used in checking real-time properties. Events may additionally have associated numerical values to facilitate program checking by the monitor. Although it is conceivable to combine the event recognizer with the filter, we chose to separate them to provide flexibility in an implementation of the framework.

Run-time Checker. The run-time checker determines whether or not the current execution history satisfies the given requirements (written MEDL). The execution history is captured from a sequence of events sent by the event recognizer. The checker can handle behavioral as well as numerical requirements. The latter can be analyzed using the technique of program checking. It may seem that the detection of a requirement violation at run-time is too late for recovery. This, however, is not necessarily true. A monitored property may represent a potentially dangerous condition that needs an attention from a human operator, which is the function that the run-time checker provides.

3 The MaC Language

In this section, we give a brief overview of the languages used to describe what to observe in the program and the requirements the program must satisfy. The scripts written in these languages are then used to automatically generate the event recognizer and the run-time checker, respectively.

The language for monitoring scripts is called PEDL (Primitive Event Definition Language, Section 3.4). PEDL scripts are used to define what information is sent from the filter to the event recognizer, and how they are transformed into requirements-level events by the event recognizer. Requirement specifications are written in MEDL (Meta Event Definition Language, Section 3.5). The primary reason for having two separate languages in the monitoring framework is to separate implementation-specific details of monitoring from requirements specification. This separation ensures that the framework is scalable to different implementation languages and specification

Formally Specified Monitoring of Temporal Properties*

Moonjoo Kim, Mahesh Viswanathan,
Hanêne Ben-Abdallah†, Sampath Kannan, Insup Lee, and Oleg Sokolsky‡
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

We describe the Monitoring and Checking (MaC) framework which provides assurance on the correctness of an execution of a real-time system at run-time. Monitoring is performed based on a formal specification of system requirements. MaC bridges the gap between formal specification, which analyzes designs rather than implementations, and testing, which validates implementations but lacks formality. An important aspect of the framework is a clear separation between implementation-dependent description of monitored objects and high-level requirements specification. Another salient feature is automatic instrumentation of executable code.

The paper presents an overview of the framework, languages to express monitoring scripts and requirements, and a prototype implementation of MaC targeted at systems implemented in Java.

1 Introduction

Real-time systems often arise in the area of embedded and safety-critical applications. Dependability of such systems is the utmost concern to their developers. Much research in the past two decades concentrated on methods for analysis and validation of real-time systems. Important results have been achieved, in particular, in the area of formal verification [4]. Formal methods of system analysis allow developers to specify their systems using mathematical formalisms and prove properties of these specifications. These formal proofs increase confidence in correctness of the system's behavior.

Still, complete formal verification has not yet become a prevalent method of analysis. The reasons for this are twofold. First, full verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, verification results apply not to system implementations, but to formal specifications of these systems. Construction of such specifications is usually a manual and error-prone process. Separate methods are needed, then, to verify compliance of the system implementation to its formal specification. Testing, on the other hand, allows one to validate the system implementation directly. However, testing results lack the rigor of formal analysis and usually do not provide guarantees of absence of errors in the implementation.

Consequently, whichever analysis approach has been taken to validate a real-time system, there exists a possibility of incorrect behavior during the execution of the system. Run-time monitoring and checking strives to address this problem.

Computer systems are often monitored for performance evaluation and enhancement [10], debugging and testing [14], and to control or check of system correctness [18]. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community [3, 15, 16, 13, 17]. Such monitors can be used to detect violations of timing [13] or logical [3] properties of a program, constraints on language constructs [15], and so on.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly with respect to a formal requirements specification. The use of formal methods is the salient aspect of our approach. We concentrate on the following two issues: (1) how to map high-level abstract events that are used in requirement specification to low-level activities of a running system, and (2) how to instrument the code to extract and detect necessary low-level

activities. We assume that both requirement specifications and the system implementation are available to us.

The major phases of the framework are as follows: (1) system requirements are formalized; at the same time, a monitoring ***script*** is constructed, which is used to instrument the code and establish a mapping from low-level information into high-level events; (2) at run-time, events generated by the instrumented system are monitored for compliance with the requirements specification. The run-time monitoring and checking (MaC) architecture consists of three components: ***filter, event recognizer,*** and run-time *checker*. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented code. The filter sends this information to the event recognizer, which converts it into high-level events and conditions and passes them to the run-time checker.

Each event delivered to the checker has a timestamp, which reflects the actual time of the occurrence of the event. This enables us to monitor real-time properties of the system. Timestamps are assigned to events by the event recognizer based on the clock readings provided by the filter. The run-time checker checks the correctness of the system execution ***thus far*** according to a requirements specification of the system, based on the information it receives from the event recognizer, and on the past history. The checker can combine monitoring of behavioral correctness of the system control flow with program checking [2] for numerical computations. This integrated approach is a unique feature of the proposed framework.

The current prototype implementation of the MaC architecture, monitors systems written in Java. Instrumentation is performed automatically, directly in JAVA bytecode. A language called MEDL, based on a linear temporal logic, is used to describe the formal requirements. Other formal languages can be readily used to specify requirements.

Related work. The "behavioral abstraction" approach to monitoring was pioneered by Bates and Wileden [1]. Although their approach lacked formal foundation, it provided an impetus for future developments. Several other approaches pursue goals that are similar to ours. The work of [5] addresses monitoring of a distributed bus-based system, based on a Petri Net specification. Since only the bus activity is monitored, there is no need for instrumentation of the system. The authors of [16] also consider only input/output behavior of the system. In our opinion, instrumentation of key points in the system allows us to detect violations faster and more reliably, without sacrificing too much performance. The test automation approach of [14] is also targeted towards monitoring of black-box systems without resorting to instrumentation. Additionally, we aim at using the MaC framework beyond testing, during real system executions. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [15]. The user manually annotates an Ada program with constructs from ANNA, a formal specification language. Mok and Liu [13] proposed an approach for monitoring the violation of timing constraints written in the specification language based on Real-time Logic as early as possible with low-overhead. The framework proposed in this paper does not limit itself to any particular kind of monitored properties. In [10], an elaborate language for specification of monitored events based on relational algebra is proposed. Instrumentation of high-level source code is provided automatically. Collected data are stored in a database. Since the instrumentation code performs database queries, instrumentation can significantly alter the performance of a program.

The paper is organized as follows. Section 2 presents an overview of the framework. Section 3 informally presents the language for monitoring scripts and requirements specifications. Section 4 describes a prototype implementation of the MaC framework. More complete and formal treatment of MaC is given in [9].

2 Overview of the MaC Framework

The MaC framework aims at run-time assurance monitoring of real-time systems. The structure of the framework is shown in Figure 1. The framework includes two main phases: (1) before the system is run, its implementation and requirement specification are used to generate run-time monitoring components; (2) during system execution, information about the running system is collected and matched against the requirements.

A major task during the first phase (indicated by clear boxes in Figure 1) is to provide a mapping between high-level events used in the requirement specification, and low-level state information extracted during execution. They are related explicitly by means of a ***monitoring script.*** The

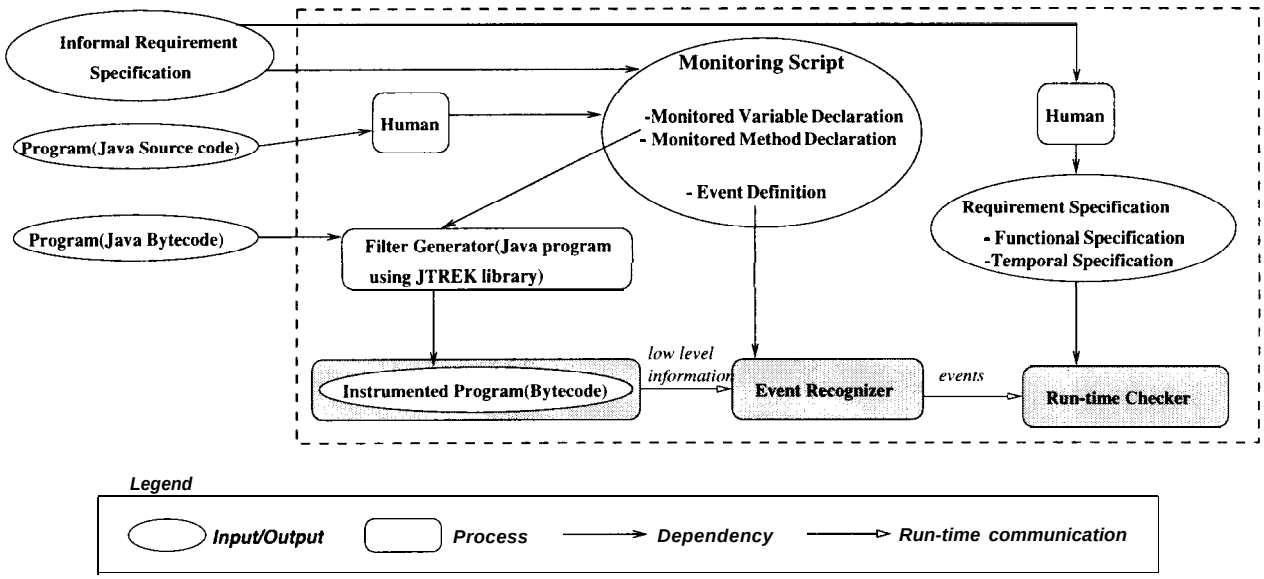


Figure 1. Overview of the MaC framework

monitoring script describes how events at the requirements level are defined in terms of monitored states of an implementation. For example, in a gate controller of a railroad crossing system, the requirements may be expressed in terms of the event train-in-crossing. The implementation, on the other hand, stores the train's position with respect to the crossing in a variable train-position. The monitoring script in this case can define the event as condition train-position < 800. The language of monitoring scripts event recognizer (described in Section 3) has limited expressive power in order to ensure fast recognition of events.

The monitoring script is used to generate a *filter* and an ***event recognizer*** automatically. The filter instruments the implementation to extract the necessary state information at run-time. The event recognizer receives state information from the filter and determines the occurrences of events according to their definition in the script. Also during the first phase, the system requirements are formalized, and a run-time ***checker*** is produced from the formal requirements. The requirement specification uses events defined in the monitoring script.

During the run-time phase (shaded boxes in Figure 1), the instrumented implementation is executed while being monitored and checked against the requirements specification. The filter sends relevant state information to the event recognizer, which determines the occurrence of events. These events are then relayed to the run-time checker to check adherence to the requirements.

Filter. A filter is a set of program fragments that are inserted into the implementation to instrument the system. The essential functionality of a filter is to keep track of changes to monitored objects and send pertinent state information to the event recognizer. Instrumentation is performed statically directly on the executable code (bytecode, in the case of Java). Instrumentation is automatic, which is made possible by the low-level description in the monitoring script.

Event recognizer. The event recognizer is the part of the monitor that detects an event from values of monitored variables received from the filter according to the monitoring script. Recognized events are delivered to the run-time checker. Each event is supplied with a timestamp that can be used in checking real-time properties. Events may additionally have associated numerical values to facilitate program checking by the monitor.

While it is conceivable to merge the event recognizer with the filter, we chose to separate the two modules. The separation allows us to remove the overhead of abstracting out events from the low-level information. This reduces interference of the monitor with the monitored system's execution. On the other hand, communication overhead incurred by sending changes in the monitored data from the filter to the event recognizer increases, but it applies only to the off-line processing of the monitored information and is therefore more acceptable. An additional advantage of the chosen design is a clear separation of monitoring activity from the system

MaC: A Framework for Run-time Correctness Assurance of Real-Time Systems *

Moonjoo Kim, Mahesh Viswanathan
Hanêne Ben-Abdallah†, Sampath Kannan, Insup Lee, and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA

January 29, 1999

*This research was supported in part by NSF CCR-9619910, AFOSR F49620-95-1-0508, ARO DAAG55-98-l-0393, ARO DAAG55-98-1-0466, and ONR N00014-97-1-0505 (MURI)

†Visiting from Département d'Informatique, FSEG, Universitk de Sfax, Tunisia

Abstract

We describe the Monitoring and Checking (MaC) framework which provides assurance on the correctness of program execution at run-time. Our approach complements the two traditional approaches for ensuring that a system is correct, namely *static analysis* and testing. Unlike these approaches, which try to ensure that all possible executions of the system are correct, our approach concentrates on the correctness of the current execution of the system.

The MaC architecture consists of three components: a filter, an event recognizer, and a run-time checker. The ***filter*** extracts ***low-level information, e.g.,*** values of program variables and function calls, from the system code, and sends it to the ***event recognizer.*** From this low-level information, the event recognizer detects the occurrence of "abstract" requirements-level ***events,*** and informs the ***run-time*** checker about them. The run-time checker uses these events to check that the current system execution conforms to the formal requirements specification of the system.

This paper overviews our current prototype implementation, which uses JAVA as the implementation language and our Monitoring Script language as the requirements language.

1 Introduction

We develop a framework for run-time monitoring of correctness of real-time systems based on a formal specification of system requirements. Computer systems are often monitored for performance evaluation and enhancement, debugging and testing, control or check of system correctness [Sch95]. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community [CG92, SM93, ML97, Sch98]. Such monitors can be used to detect violations of timing [ML97], a logical property of a program [CG92], a constraint on a language construct [SM93], and so on.

The reason for increased interest in correctness monitors is that it is becoming more difficult to test or verify software because software size is increasing and its functionality is becoming more complicated. The most common way to validate a software system is testing. However, testing cannot be used to guarantee that the system is error-free, since it is infeasible to completely test the entire system due to the large number of possible behaviors. Also, as the functionality and structure of software becomes complex in order to satisfy a broad range of needs, testing itself needs to be sophisticated enough to check the program according to diverse criteria. For example, for testing a numerical computation, it is enough to check output with given input. However, when we test a real-time application like traffic control system, we also have to check the timing behavior.

Formal verfication has been used to increase the confidence that a system will be correct by making sure that a design specification is correct. However, even if a design has been formally verified, it still does not ensure the correctness of an implementation of the design. This is because the implementation often is much more detailed, and may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of a design that has been verified. One way to overcome this gap between the design and the implementation is to resort to testing the implementation's behavior on a set of

input sequences derived from the specification. This approach, however, suffers from the same drawback as testing in general and does not provide guarantees about the correctness of the implementation on *all* possible input sequences. Consequently, we cannot guarantee, using the two traditional methods prior to the execution of the system, that its run-time behavior will be correct. Therefore, the approach of continuously monitoring a running system has received much attention.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly. The salient aspect of our approach is the use of formal requirements specification to decide what properties to assure. Since our goal is to check an implementation against requirements specification at run-time, we assume that we are given both requirement specifications and an implementation. To be able to monitor satisfaction of requirements, we have to correlate low-level observations with high-level notions used in the requirements specification. Therefore, the primary concern of our presentation are the following two issues:

- how to map high-level abstract events that are used in a requirement specification to low-level activities of a running system
- how to instrument code to extract necessary low-level activities.

The framework consists of the three phases: the design phase, the implementation and instrumentation phase, and the run-time phase. During the design phase, the requirements on the system are specified. Optionally, a formal system specification may also be written down and in this case we assume that verification is done to ensure that the system specification satisfies the requirements. During the implementation phase the system is implemented. Based on the requirements specification and the implementation, the user provides a monitoring *script* that contains instructions for instrumenting the code so that low-level information about program state can be passed on to the monitor. In addition, it

Graphical Cornrnunicating Shared Resources: a Language for Specifying and Analyzing Real-The System

Hanêne Ben-Abdallah
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3Gl
hanene@swen.uwaterloo.ca

Insup Lee
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
lee@central.cis.upenn.edu

January 1997

Abstract

We present Graphical Communicating Shared Resources, GCSR, a formal language for the specification and analysis of real-time systems, including their functional, temporal and resource requirements. GCSR supports the explicit representation of system resources and priorities to arbitrate resource contentions. These features allow a designer to examine resource inherent constraints and to experiment with various resource allocations and scheduling disciplines in order to produce a more dependable specification. In addition, GCSR differs from other graphical languages through its well-defined notions of modularity and hierarchy: dependencies between system components, expressed as communication events, can have a limited scope of visibility, and control flow between components is clearly represented as either an interrupt. or exception! i.e., voluntary release of control. Further, GCSR has a precise operational semantics and notions of equivalence that allow the execution and formal analysis of a specification. We present the GCSR language, its toolset, and how properties, e.g., safety can be analyzed within GCSR.

Keywords- Design specification, executable specifications, graphical languages, process algebra, real-time systems, requirements specification.

Contents

1 Introduction

The timed behavior of a real-time system is affected not only by the time its components take to execute and synchronize, but also by delays introduced due to the scheduling of tasks that compete for shared resources. Most current real-time formalisms adequately capture delays due to component synchronization, e.g., Statecharts [10], Modechart [18], Communicating Real-time State Machine [32], and timed extensions of the classic untimed process algebras CSP and CCS [8, 15, 25, 35, 27]. These formalisms, however, abstract out resource-specific details. This motivated the Communicating Shared Resources (CSR) paradigm [19, 21] to provide a formalism where the run-time resource requirements of a real-time system can be specified together with its functional and temporal requirements. The integration of the three types of requirements allows designers to consider resource-induced constraints early in the development cycle, explore alternate resource allocations, and to eliminate unimplementable design alternatives without expensive prototyping.

Within the CSR paradigm, the Algebra of Communicating Shared Resources (ACSR) [19] and the Graphical Communicating Shared Resources (GCSR) [4, 1] have been developed. ACSR is a timed process algebra and GCSR is a graphical language. The novelty of these formalisms relative to existing real-time formalisms is their representation of resources and priority. Without an explicit notion of resources, the specification of resource-bound systems requires that some artificial means be used to model resource requirements. such as defining processes to represent resources. Models that lack explicit priorities require that a process be created for the sole purpose of arbitrating priorities and implementing preemption. Providing explicit notions of resources and priority within the CSR formalisms results in specifications that are close analogues of the systems they model and that are easier to modify to reflect different. resource allocations and priority-based scheduling disciplines.

In this paper, we present' the GCSR language and its toolset PARAGON. The development of GCSR has several motivations. One, as mentioned earlier, is to provide a formalism where the three types of requirements for real-time systems can be formally modeled and rigorously analyzed. Another motivation is to provide a pair of graphical and textual languages with a common semantics. This gives the software engineer the flexibility to mix both types of specifications. From our experience, the graphical notation better describes the high-level view of a system along with the interactions and dependencies between its various components. On the other hand, the textual notation concisely describes the details of components. We therefore ensured that both GCSR and ACSR have common semantics, which in turn made GCSR benefit from the process algebraic analysis techniques [21]: execution, testing, state space exploration, and automated equivalence checking. These analysis techniques are used to verify whether a GCSR design of a system satisfies its requirements. In addition, the equivalence relations of ACSR allows a designer to restructure a GCSR specification, e.g., minimize its size, without affecting its behavior. The congruence equivalence relations of ACSR allows a designer to replace one specification with an equivalent one inside a larger system. This in turn makes modular specification and verification possible within the GCSR formalism.

A third motivation for GCSR is to support the modular, hierarchical and thus scalable specifications of real-time systems. The graphical syntax of GCSR adopts the intuitive notions of edges and nodes in control flow diagrams and provides various types of node and edge for scalability. In particular, the visibility scope of communication events, which are used to describe dependencies between components, can be limited inside node boundaries. Further, GCSR disallows edges from crossing node boundaries and offers instead two types of edges that graphically distinguish two types of control exits from inside a node: via interrupt versus exception, i.e., voluntary release of control.

The PARAGON toolset has been vital in our experiments with the expressiveness of GCSR and ACSR, as well as in our evaluation of the benefits of integrating the resource requirements with the functional and temporal requirements. We used PARAGON to model and analyze several case studies, e.g., the production cell [23, 1], scheduling problems [3, 6], avionic applications [4, 2], and various versions of the railroad crossing system benchmark [14, 1. 21]. In this paper, we illustrate the GCSR language and analysis techniques applied to parts of the production cell case study. This example represents a. realistic, industrial real-time application, where safety requirements are essential. Whether or not these safety requirements are satisfied can be shown by the application of a formal method. For this application, the GCSR modularity allowed us to model distinctively the two main agents in the system: the environment (i.e., the cell's physical machines) and the software system (i.e., software controllers) [36]. The graphical syntax of GCSR facilitated the visualization of the system components and their resource and communication dependencies. In addition, the explicit specification of resources allowed us to express in a natural way the violations of safety requirements pertinent to system resources. Further, the analysis techniques provided within the CSR paradigm were suitable for verifying that our solution satisfies all of the requirements.

Paper organization. The next section overviews the GCSR language. Section 3 defines the GCSR syntax and semantics. Section 4 describes the PARAGON toolset. Section 5 informally describes the production cell as reported in [23] and presents parts of our design solution and analysis in GCSR. Section 6 reviews relevant' work. Finally, Section 7 summarizes the main features of GCSR. and outlines future research.

2 An overview

The GCSR paradigm is based on the view that a real-time system consists of a set of communicating components, called processes, that execute on a finite set of serial resources and synchronize with one another through communication channels. The use of shared resources is represented by timed actions, and synchronization is supported by instantaneous events. The execution of an action takes nonzero time units with respect to a global clock, and consumes a set of resources during that time. The execution of an action is subject to the availability of the resources it uses. Contention for resources is arbitrated according to

Parametric Approach to the Specification and Analysis of Real-time System Designs based on ACSR-VP *

Hee-Hwan Kwak, Insup Lee, and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA
lee@cis.upenn.edu, {heekwak,sokolsky}@saul.cis.upenn.edu

July 19, 1999

Abstract

To engineer reliable real-time systems, it is desirable to discover timing anomalies early in the development process. However, there is little work addressing the problem of accurately predicting timing properties of real-time systems before implementations are developed. This paper describes an approach to the specification and analysis of scheduling problems of real-time systems. The method is based on ACSR-VP, which is an extension of ACSR, a real-time process algebra, with value-passing capabilities. Combined with the existing features of ACSR for representing time, synchronization and resource requirements, ACSR-VP can be used to describe an instance of a scheduling problem as a process that has parameters of the problem as free variables. The specification is analyzed by means of a symbolic algorithm. The outcome of the analysis is a set of equations and a solution to which yields the values of the parameters that make the system schedulable. These equations can be solved using integer programming or constraint logic programming. The paper presents the theory of ACSR-VP briefly and an example of the period assignment problem for rate-monotonic scheduling. We also explain our current tool implementation effort and plan for incorporating it into the existing toolset, PARAGON.

1 Introduction

The desire to automate or incorporate intelligent controllers into control systems has lead to rapid growth in the demand for real-time software systems. Moreover, these systems are becoming increasingly complex and require careful design analysis to ensure reliability before implementation. Recently, there has been much work on formal methods for the specification and analysis of real-time systems [8, 12]. Most of the work assumes that various real-time systems attributes, such as execution time, release time, priorities, etc., are fixed ***a priori*** and the goal is to determine whether a system with all these known attributes would meet required safety properties. One example of safety property is schedulability analysis; that is, to determine whether or not a given set of real-time tasks under a particular scheduling discipline can meet all of its timing constraints.

The pioneering work by Liu and Layland [17] derives schedulability conditions for rate-monotonic scheduling and earliest-deadline-first scheduling. Since then, much work on schedulability analysis has been done which includes various extensions of these results [11, 28, 25, 4, 26, 22, 18, 3]. Each of these extensions expands the applicability of schedulability analysis to real-time task models with different assumptions. In particular, there has been much advance in scheduling theory to address uncertain nature of timing attributes at the design phase of a real-time system. This problem is complicated because it is not sufficient to consider the worst case timing values for schedulability analysis. For example, scheduling anomalies can occur even when there is only one processor and jobs have variable execution times and are nonpreemptable. Also for preemptable jobs with one processor, scheduling anomalies can occur when jobs have arbitrary release times and share resources. These scheduling anomalies make the problem of validating a priority-driven system difficult. Clearly, exhaustive simulation or testing is not practical in general except for small systems of practical interest. There have been many different heuristics developed to solve some of these general schedulability analysis problems. However, each algorithm is problem specific and thus when a problem is modified, one has to develop new heuristics.

*This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-I-0466, AFOSR F49620-96-I-0204, NSF CCR-96 199 IO, and ONR N00014-97-1-0505.

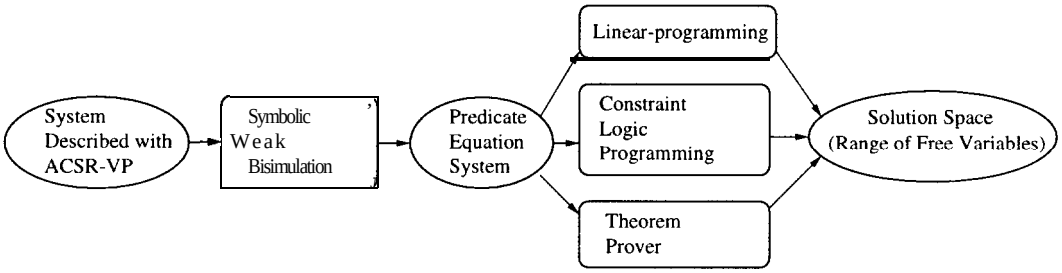


Figure 1: Overview of the Framework

In this paper, we describe a framework that allows one to model scheduling analysis problems with variable release and execution times, relative timing constraints, precedence relations, dynamic priorities, multiprocessors etc. Our approach is based on ACSR-VP and symbolic bisimulation algorithm.

ACSR (Algebra of Communicating Shared Resources) [14], is a discrete real-time process algebra. ACSR has several notions, such as resources, static priorities, exceptions, and interrupts, which are essential in modeling real-time systems. ACSR-VP is an extension of ACSR with value-passing and parameterized processes to be able to model real-time systems with variable timing attributes and dynamic priorities. In addition, symbolic bisimulation for ACSR-VP has been defined. ACSR-VP without symbolic bisimulation has been applied to the simple schedulabihty analysis problem [5], by assuming that all parameters are ground, i.e., constants. However, it is not possible to use the technique described in [5] to solve the general schedulabihty analysis problem with unknown timing parameters.

Figure 1 shows the overall structure of our approach. We specify a real-time system with unknown timing or priority parameters in ACSR-VP. For the schedulability analysis of the specified system, we check symbolically whether or not it is bisimilar to a process idling forever. The result is a set of predicate equations, which can be solved using widely available linear-programming or constraint-programming techniques. The solution to the set of equations identifies, if exists, under what values of unknown parameters the system becomes schedulable. To support the effective use of the the symbolic ACSR-VP analysis, we are developing a tool and planning to integrate into PARAGON [27], a toolset with graphical interface to support the use of ACSR.

The rest of the paper is organized as follows. Sections 2 overviews the theory of the underlying formal method, ACSR-VP, and introduce symbolic bisimulation for ACSR-VP expressions. Section 3 gives a specification of a scheduling problem, namely the ***period assignment problem*** and illustrates how to analyze an instances of this problem. Section 4 briefly describes the PARAGON toolset and its support for value-passing specifications, and outlines the incorporation of ACSR-VP into the toolset. We conclude with a summary and an outline of future work in Section 5.

2 ACSR-VP

ACSR-VP extends the process algebra ACSR [14] by allowing values to be communicated along communication channels. In this section we present ACSR-VP concentrating on its value-passing capabilities. We refer to the above papers for additional information on ACSR.

We assume a set of variables X ranged over by x, y, a set of values ***V*** ranged over by v, and a set of labels ***L*** ranged over by c, d. Moreover, we assume a set $Expr$ of expressions (which includes arithmetic expressions) and we let $BExpr$ c ***Expr*** be the subset containing boolean expressions. We let e and b range over $Expr$ and $BExpr$ respectively, and we write $\vec{z}$ for a tuple $z_1, \ldots z_n$ of syntactic entities.

ACSR-VP has two types of actions: instantaneous communication and timed resource access. Access to resources and communication channels is governed by priorities. A priority expression p is attached to every communication event and resource access. A partial order on the set of events and actions, the preemption relation, allows one to model preemption of lower-priority activities by higher-priority ones.

Instantaneous actions, called ***events,*** provide the basic synchronization and communication primitives in the process algebra. An event is denoted as a pair (i, e_p) representing execution of action i at priority e_p, where i ranges over τ, the idle action, $c?x$, the input action, and $c!e$, the output action. We use $\mathcal{D}_E$ to denote the domain of events and let λ range over events. We use I(X) and $\pi(\lambda)$ to represent the label and priority, respectively, of the event λ; e.g., $l((c!x, p))$ = c! and $l((c?x, p))$ = ***c?.*** To model resource access, we assume that a system contains a finite set of serially-reusable resources drawn from some set ***R.*** An action that consumes one tick of time is drawn from the domain ***P(R*** x ***Expr)*** with the restriction that each resource is represented at most once. For example the singleton action $\{(r, e_p)\}$ denotes the use of some resource $r \in$ ***R*** at priority

level e_p. The action $\emptyset$ represents idling for one unit of time, since no resource is consumed. We let $\mathcal{D}_R$ to denote the domain of timed actions with **A, B,** to range over $\mathcal{D}_R$. We define **p(A)** to be the set of the resources used by action **A,** for example $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the use of the resource r in the action A; e.g., $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$, and write $\pi_r(A) = 0$ if $r \notin$ **p(A).** The entire domain of actions is denoted by $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α, β range over $\mathcal{D}$. We let P, **Q** range over ACSR-VP processes and we assume a set of process constants ranged over by C. The following grammar describes the syntax of ACSR-VP processes:

$$P ::= \text{NIL} \mid A : P \mid \lambda.P \mid P + P \mid P \| P \mid b \rightarrow P \mid P \backslash F \mid [P]_I \mid C(\vec{x}).$$

In the input-prefixed process $(c?x, e).P$ the occurrences of variable x is bound. We write **fv(P)** for the set of free variables of **P.** Each agent constant C has an associated definition $C(\vec{Z}) \stackrel{\text{def}}{=} P$ where **fv(P)** $\subseteq \vec{x}$ and $\vec{x}$ are pairwise distinct. We note that in an input prefix $(c?x, e).P$, e should not contain the bound variable x, although x may occur in **P.**

An informal explanation of ACSR-VP constructs follows: The process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The first, $A : P$, executes a resource-consuming action during the first time unit and proceeds to process **P.** On the other hand $\lambda.P$, executes the instantaneous event λ and proceeds to **P.** The process $P + Q$ represents a nondeterministic choice between the two summands. The process $P \| Q$ describes the concurrent composition of **P** and Q: the component processes may proceed independently or interact with one another while executing instantaneous events, and they synchronize on timed actions. Process $b \rightarrow P$ represents the conditional process: it performs as **P** if boolean expression **b** evaluates to ***true*** and as NIL otherwise. In $P \backslash F$, where $F \subseteq L$, the scope of labels in **F** is restricted to process **P:** components of **P** may use these labels to interact with one another but not with **P**'s environment. The construct $[P]_I, I \subseteq R$, produces a process that reserves the use of resources in I for itself, extending every action 4 in **P** with resources in $I - \rho(A)$ at priority 0.

The semantics of ACSR-VP processes may be provided as a labeled transition system, similarly to that of ACSR. It additionally makes use of the following ideas: Process $(c!e_1, e_2).P$ transmits the value obtained by evaluating expression e_1 along channel c, with priority the value of expression e_2, and then behaves like **P.** Process $(c?x, p).P$ receives a value v from communication channel c and then behaves like $P[v/x]$, that is **P** with v substituted for variable x. In the concurrent composition $(c?x, p_1).P_1 \| (c!v, p_2).P_2$, the two components of the parallel composition may synchronize with each other on channel c resulting in the transmission of value v and producing an event $(\tau, p_1 + p_2)$.

2.1 Unprioritized Symbolic Graphs with Assignment

Consider the simple ACSR-VP process $P \stackrel{\text{def}}{=} (in?x, 1).(out!x, 1).\text{NIL}$ that receives a value along channel in and then outputs it on channel *out*, and where x ranges over integers. According to traditional methods for providing semantic models for concurrent processes, using transition graphs, process **P** in infinite branching, as it can engage in the transition $(in?n, 1)$ for every integer n. As a result standard techniques for analysis and verification cannot be applied to such processes.

Several approaches have been proposed to deal with this problem for various subclasses of value-passing processes [9, 16, 20, 13]. One of these advocates the use of *symbolic* semantics for providing finite representations of value-passing processes. This is achieved by taking a more conceptual view of value-passing than the one employed above. More specifically consider again process **P.** A description of its behavior can be sufficiently captured by exactly two actions: an input of an integer followed by the ouput of this integer. Based on this idea the notion of symbolic transition graph [9] and transition graphs with assignment [16] were proposed and shown to capture a considerable class of processes.

In this section we present symbolic graphs with assignment for ACSR-VP processes. As it is not the intention of the paper to present in detail the process-calculus theory of this work, we only give an overview of the model and we refer to [13] for a complete discussion.

2.2 Symbolic Graph with Assignment

The notion of a ***substitution,*** which ***we*** also call ***assignment,*** is defined as follows. A ***substitution*** is any function $\theta : X \rightarrow Expr$, such that $\theta(x) \neq$ x for a finite number of x$\in X$. Given a substitution θ, the ***support*** (or ***domain)*** of θ is the set of variables $D(\theta) = \{\text{x} \mid \theta(x) \neq \text{x}\}$. A substitution whose support is empty is called the ***identity substitution,*** and is denoted by Id. When $|D(\theta)| = 1$, we use $[\theta(x)/x]$ for the substitution 0. Given two substitutions θ and σ, the ***composition*** of θ and σ is the substitution denoted by $\theta;\sigma$ such that for every variable x, $\theta;\sigma(x) = \sigma(\theta(x))$. We often write $\theta\sigma$ for $\theta;\sigma$.

An SGA is a rooted directed graph where each node n has an associated finite set of free variables **fv**(n) and each edge is labeled by a guarded action with assignment [16,23]. Note that a node in SGA is a ACSR-VP term.

Definition 2.1 (SGA) A Symbolic Graph with Assignment (SGA) for ACSR-VP is a rooted directed graph where each node n has an associated ACSR-VP term and each edge is labeled by boolean, action, assignment (b, α, θ). □

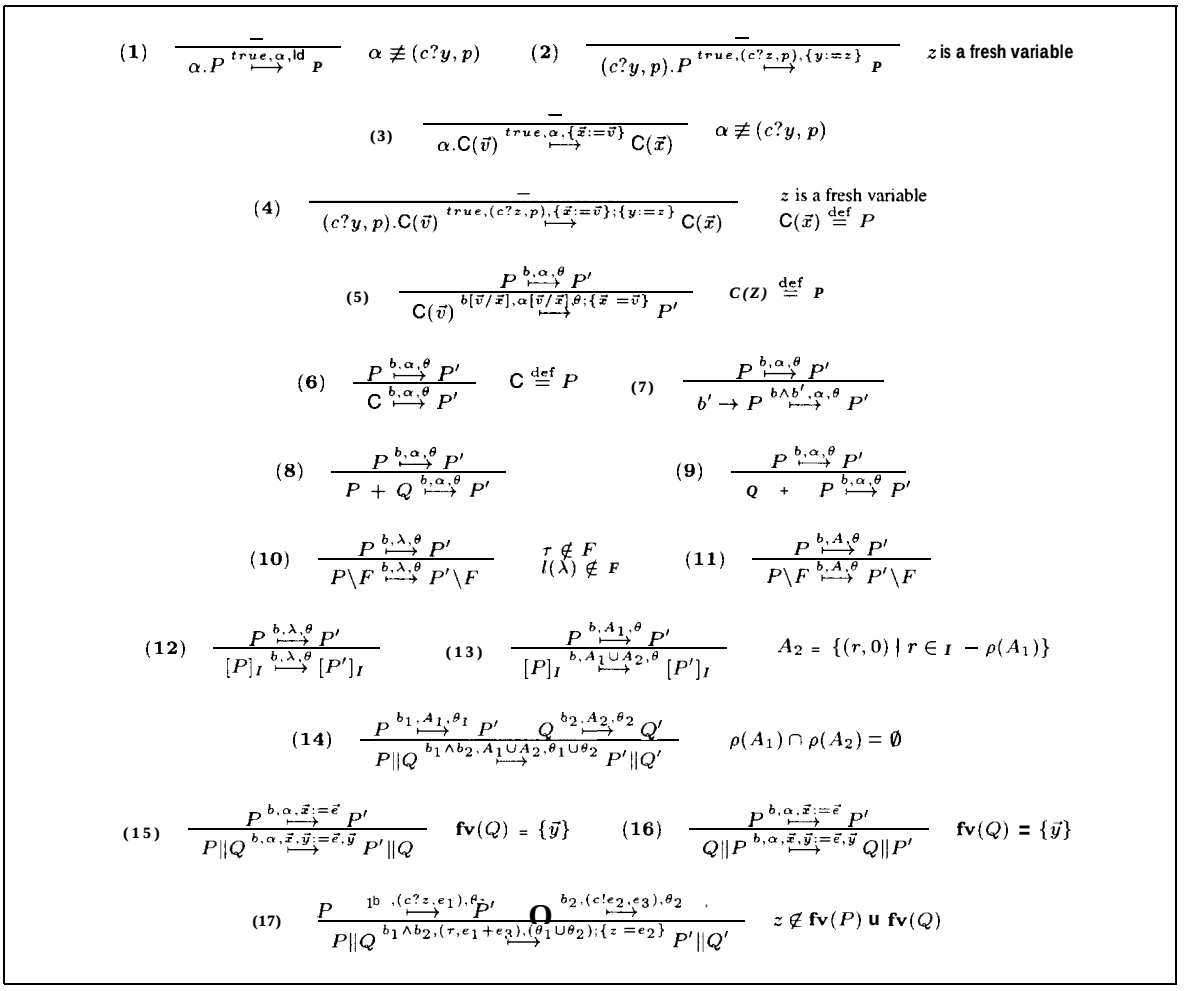
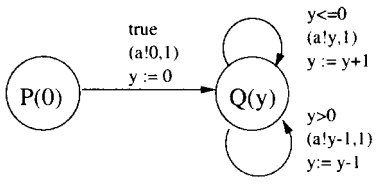


Figure 2: Rules for constructing Symbolic Graphs with Assignment

Given an ACSR-VP term, a SGA can be generated using the rules in Figure 2. Transition $P \overset{b,\alpha,\theta}{\longmapsto} P'$ denotes that given the truth of boolean expression b, P can evolve to P' by performing actions α and putting into effect the assignment 8. The interpretation of these rules is straightforward and we explain them by an example: Consider the following process. Process $P(0)$ can output the sequence of events $a!0$ infinitely many times.

$$P(x) \overset{\mathrm{def}}{=} (a!x, 1).Q(x)$$
$$Q(y) \overset{\mathrm{def}}{=} (y \leq 0) \to (a!y, 1).Q(y+1) + (y > 0) \to (a!y-1, 1).Q(y-1)$$

Following SGA represents the process $P(0)$.



Symbolic Weak Bisimulation for Value-passing Calculi *

Hee-Hwan Kwak, Jin-Young Choi†, Insup Lee, Anna Philippou
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{heekwak,annap}@saul.cis.upenn.edu, lee@cis.upenn.edu
† Department of Computer Science and Engineering
Korea University, Seoul, Korea
choi@formal.korea.ac.kr

Abstract

Bisimulation equivalence has proved to be a useful notion for providing semantics to process calculi, and it has been an object of rigorous study within the concurrency theory community. Recently there have been significant efforts for extending bisimulation techniques to encompass process calculi that allow the communication of data, that is value-passing calculi. A main challenge in achieving this has been the fact that in value-passing processes, where the values are from an infinite domain, checking for process equivalence involves comparing an infinite number of behaviors. Central among work in this area is the approach in which a new notion of process semantics, namely ***symbolic semantics,*** is proposed as a means of expressing a class of value-passing processes as finite graphs, although their traditional underlying graphs are infinite. This paper presents new algorithms for computing symbolic weak bisimulation for value-passing processes. These algorithms are built on the basis of a new formulation of late weak bisimulation which we propose. Unlike the traditional definition, this formulation allows the use of a single symbolic weak transition relation for expressing both early and late bisimulations. In addition to being easy to implement, our algorithms highlight the distinction between early and late weak bisimulations. Furthermore, we propose a simple variation of symbolic transition graphs with assignment proposed by Lin, in which the order of assignments and actions in transitions is exchanged. We believe that the resulting graphs are easier to construct and more intuitive to understand.

*This research was supported in part by NSF CCR-9415346, NSF CCR-9619910, AFOSR F49620-95-1-0508, ARO DAAH04-95-1-0092, and ONR N00014-97-1-0505.

1 Introduction

Bisimulation equivalence [4] has proved to be a useful notion for providing semantics to process calculi, and it has been an object of rigorous study within the concurrency theory community. As a result, today the theory of bisimulation is well understood within the context of traditional process calculi such as CCS, and efficient bisimulation-checking algorithms exist that enable its automatic checking.

Recently there have been significant efforts for extending bisimulation techniques to encompass process calculi that allow the communication of data, that is value-passing calculi [4, 5, 3]. A main challenge in achieving this has been the fact that in value-passing processes, where the values are from an infinite domain, checking for process equivalence involves comparing an infinite number of behaviors. A substantial body of work has concentrated on addressing this challenge and several approaches have been proposed for establishing bisimulation for classes of value-passing processes. Central among them is the approach proposed by [1] where a new notion of process semantics, namely *symbolic semantics*, is proposed as a means of expressing a class of value-passing processes as finite graphs, although their traditional underlying graphs are infinite. Strong bisimulation equivalence was redefined at this level and algorithms for deciding it were presented. The class of processes that can be described by finite symbolic graphs was extended in [3], which generalized the notion of symbolic transition graphs by allowing assignments to take place in transitions, yielding *symbolic transition graphs with* assignment, STGA's in short, and the theory was appropriately extended. We refer to these papers for a detailed exposition. In this setting, bisimulation for symbolic transition graphs is defined in terms of relations parametrized on boolean expressions, of the form $\simeq^b$, where $p \simeq^b q$ if and only if, for each interpretation satisfying boolean b, p and q are bisimilar in the traditional notion. Furthermore, algorithms were presented for computing these equivalences. In particular, given two closed processes whose symbolic transition graphs are finite, the algorithm constructs a predicate equation system that corresponds to the most general condition for the two processes to be bisimilar.

An additional particularity of value-passing processes is the existence of more than one way for providing their semantics: as noted and investigated in the context of the π-calculus [5], a process-calculus where names, or channels are passed as objects of interaction among processes, there are two ways of interpreting input actions. These are often referred to as early and *late* semantics, which distinguish on the time when the receipt of a value takes place during an input transition. Each of these semantics, gives rise to the corresponding bisimulation equivalence, and it turns out that *late* bisimulation is strictly finer than early bisimulation. The work of [1, 3] deals with both of these equivalences. We should also mention that a similar approach for deciding strong early bisimulation was independently presented in [7] and we refer to these papers for further discussion of research in this field.

This line of work has been the main motivation of this paper, our aim being to extend reasoning about SGA's to handle weak behavioral relations. Definitions of such relations were provided in [6] in the context of the π-calculus and in [8] for value-passing CCS, the latter including symbolic variants. To the best of our knowledge, work involving these equicalences has mainly focused on their characterization [9].

The main contributions of this paper can be summarized as follows: First we develop and present algorithms for computing weak bisimulation for value-passing processes, corresponding to finite SGA's. Secondly, we study in some detail the distinction between early weak and late weak bisimulations. We observe that a close relationship can be established between the two

and we present a new formulation of the late weak bisimulation definition that brings forward a new treatment of late bisimulation as a special case of early bisimulation. We use the new formulations as a basis for constructing our algorithms. An advantage of this approach is that the resulting algorithms follow easily from the definitions and the algorithm for late weak bisimulation is a simple extension of the early weak bisimulation one. As in the case of strong bisimulations, our algorithms produce logical formulas that capture the most general condition for two processes to be weakly bisimilar. Finally, in this paper we propose a simple variation of the SGA's proposed in [3], in which the order of assignments and actions in transitions is exchanged. We believe that the resulting graphs are easier to construct and more intuitive to understand. This is discussed in detail in Section 2. Similar graphs are also employed in [7], although no rules were defined for their construction in that paper.

In this paper we focus on value-passing CCS, which we refer to as CCS-VP, as a vehicle of presenting our results. However, the approach is applicable to other value-passing process calculi.

The rest of the paper is organized as follows: In the next section we present some background material on value-passing CCS and we define rules for constructing SGA's. In Section 3, we proceed to consider how operational semantics and ground bisimulations can be defined for SGA's. In Section 4, we define weak symbolic bisimulations and in Section 5 we present algorithms for computing them. We conclude with a summary and some final remarks in Section 6. An appendix contains the proof of a main result.

2 Symbolic Graph with Assignment for CCS-VP

The syntax of value-passing CCS is similar to that of CCS. The difference is that values may be passed along communication channels, and assigned to variables. We assume a set of variables X and let x, y range over X. A process P with free variables $\vec{x} = x_1, \ldots, x_n$ is denoted as P(Z). Furthermore, we use v for value constants ranging over some set *V*, *e* for value expressions ranging over some set *Expr* (including for example arithmetic expressions), and b for boolean expressions ranging over some set $BExpr$, with *Expr* $\supset$ *BExpr*. We denote a vector of value variables and value expressions as $\vec{x}$ and $\vec{e}$, respectively. Also, we assume a set of labels *L* and we use c to denote labels and F to represent subsets of *L*. Furthermore, we use C, X to stand for process constants. The following grammar describes the syntax of CCS-VP processes:

$$
\begin{array}{lcl}
\text{P} & ::= & \text{NIL} \mid \alpha.P \mid P + P \mid P \| P \mid b \rightarrow P \mid P \backslash F \mid C \\
\alpha & ::= & \tau \mid c?x \mid c!e \\
c & ::= & x \mid \text{X}(\vec{e})
\end{array}
$$

For each process constant X, we assume a declaration of the form $\text{X(Z)} \stackrel{\text{def}}{=} P$ where the free variables of P all occur in $\vec{x}$. We use $Act = \{c?x, \text{c!e} \mid \text{c} \in L\} \cup \{\tau\}$ to denote the domain of actions, and we let α, β range over *Act*. We use $l(\alpha)$ to represent the label of action α; e.g., $l(c!e) = \text{c!}$ and $l(c?x) = \text{c?}$. Lastly, given a process *P, we* write $fv(P)$ for the free variables occurring in *P* which are defined in the usual way, that is $fv(c!e.P) = fv(e) \cup fv(P)$ and $fv(c?x.P) = fv(P) - \{x\}$.

2.1 Symbolic Transition System

The Symbolic Graph with Assignment (SGA) we propose here has been used in [7]. However, no rules to construct the SGA were presented in that paper. In this Section, we introduce rules to construct the SGA from CCS-VP terms.

The notion of a *substitution,* which we also call *assignment,* is defined as follows. A *substitution* is any function $\theta: X \rightarrow Expr$, such that $\theta(x) \neq x$ for a finite number of $x \in X$. Given a substitution θ, the *support* (or *domain)* of θ is the set of variables $D(\theta) = \{x \mid \theta(x) \neq x\}$. A substitution whose support is empty is called the *identity substitution,* and is denoted by Id. When $|D(\theta)| = 1$, we use $[\theta(x)/x]$ for the substitution θ. Given two substitutions θ and σ, the *composition* of θ and σ is the substitution denoted by $\theta;\sigma$ such that for every variable x, $\theta;\sigma(x) = \sigma(\theta(x))$. $\theta\sigma$ for $\theta;\sigma$.

An SGA is a rooted directed graph where each node n has an associated finite set of free variables $fv(n)$ and each edge is labeled by a guarded action with assignment [3, 8]. Note that a node in SGA is a CCS-VP term. Furthermore, we use $\sqcup$ to denote the empty action the purpose of which is explained later.

Definition 2.1 **(SGA)** A Symbolic Graph with Assignment (SGA) for CCS-VP is a rooted directed graph where each node ***n*** has an associated CCS-VP term and each edge is labeled by boolean, action, assignment, (b, α, θ) or by boolean, the empty action and assignment, (b, $\sqcup$, θ). □

$$(1)\ \frac{}{\alpha.P \overset{true,\alpha,\mathrm{Id}}{\longmapsto} P} \qquad (2)\ \frac{C(\vec{e}) \overset{b,\sqcup,\theta}{\longmapsto} C}{\alpha.C(\vec{e}) \overset{b,\alpha,\theta}{\longmapsto} c}$$

$$(3)\ \frac{}{C(\vec{e}) \overset{true,\sqcup,\vec{x}:=\vec{e}}{\longmapsto} C} \quad C(\vec{x}) \overset{\mathrm{def}}{=} P \qquad (4)\ \frac{P \overset{b,\alpha,\theta}{\longmapsto} P'}{C \overset{b,\alpha,\theta}{\longmapsto} P'} \quad C(Z) \overset{\mathrm{def}}{=} P$$

$$(5)\ \frac{P \overset{b,\alpha,\theta}{\longmapsto} P'}{b' \rightarrow P \overset{b \wedge b',\alpha,\theta}{\longmapsto} P'}$$

$$(6)\ \frac{P \overset{b,\alpha,\theta}{\longmapsto} P'}{P + Q \overset{b,\alpha,\theta}{\longmapsto} P'} \qquad (7)\ \frac{P \overset{b,\alpha,\theta}{\longmapsto} P'}{Q + P \overset{b,\alpha,\theta}{\longmapsto} P'}$$

$$(8)\ \frac{P \overset{b,\alpha,\theta}{\longmapsto} P'}{P\backslash F \overset{b,\alpha,\theta}{\longmapsto} P'\backslash F} \quad \begin{array}{l} \tau \notin F \\ l(\alpha) \notin F \end{array}$$

$$(9)\ \frac{P \overset{b,\alpha,\vec{x}:=\vec{e}}{\longmapsto} P'}{P \| Q \overset{b,\alpha,\vec{x},\vec{y}:=\vec{e},\vec{y}}{\longmapsto} P' \| Q} \quad fv(Q) = \{\vec{y}\} \qquad (10)\ \frac{P \overset{b,\alpha,\vec{x}:=\vec{e}}{\longmapsto} P'}{Q \| P \overset{b,\alpha,\vec{y},\vec{y}:=\vec{e},\vec{y}}{\longmapsto} Q \| P'} \quad fv(Q) = \{\vec{y}\}$$

$$(11)\ \frac{P \overset{b_1,c?z,\theta_1}{\longmapsto} P' \qquad Q \overset{b_2,c!e,\theta_2}{\longmapsto} Q'}{P \| Q \overset{b_1 \wedge b_2,\tau,(\theta_1 \cup \theta_2);\{z:=e\}}{\longmapsto} P' \| Q'} \quad z \notin fv(P) \cup fv(Q)$$

Figure 1: Rules to construct Symbolic Graph with Assignment

Given an CCS-VP term, a SGA can be generated using the rules in Figure 1. Note that the purpose of action $\sqcup$ is to decorate transitions that involve no action, but are nonetheless

Part III

Result Checking

Tuğkan Batu, Ronitt Rubinfeld, and Patrick White: "Fast Approximate PCPs for Multidimensional Bin-Packing Problems", in the Proceedings of the Third International Workshop on Randomization and Approximation Techniques in Computer Science — APPROX-RANDOM'99 (D. Hochbaum, K. Jansen, J.P.D. Rolim, and A. Sinclair editors), pp. 2455256, Springer-Verlag LNCS 1671, Berkeley, CA, August 1999.

Full paper: http://www.cs.cornell.edu/Info/People/batu/m.ps

Tuğkan Batu, Ronitt *Rubinfeld, and Patrick White:* "Runtime Verification of Remotely Executed Code using Probabilistically Checkable Proof Systems", in the Proceedings of the Workshop on Run-Time Result Verification — RTRV'99, Trento, Italy, July 1999.

Full paper: http://www.cs.cornell.edu/Info/People/batu/r.ps

Funda Ergün, S. Ravi Kumar, and Ronitt Rubinfeld: "Fast approximate PCPs", in the Proceedings of the 31st ACM Symposium on Theory of Computing — STOC'99, pp. 41-50, ACM Press, Atlanta, GA, l-4 May 1999

Full paper: http://simon.cs.cornell.edu/Info/People/ronitt/PAP/ap.ps

Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan: "Spot-Checkers", to appear in the *Journal of Computer and System Sciences,* special issue on STOC'98.

Full paper: http://theory.stanford.edu/muri/funda2.ps

S. Ravi Kumar and Ronitt Rubinfeld: "Property Testing of Abelian Group Operations", manuscript.

Full paper: `http : //simon.cs.cornell.edu/Info/People/ronitt/PAP/comm.ps`

Joan Feigenbaum, Sampath Kannan, *Martin Strauss, and Mahesh Viswanathan:* "Streaming Algorithms for Distributed, Massive Data Sets", manuscript.

Full paper: `ftp://ftp.cis.upenn.edu/pub/papers/kannan/SICOMP.ps.Z`

Fast Approximate PCPs for Multidimensional Bin-Packing Problems *

Tuğkan Batu[1], Ronitt Rubinfeld[1]**, and Patrick White'

Department of Computer Science, Cornell University, Ithaca, NY 14850
{batu,ronitt,white}@cs.cornell.edu

Abstract. We consider approximate PCPs for multidimensional bin-packing problems. In particular, we show how a verifier can be quickly convinced that a set of multidimensional blocks can be packed into a small number of bins. The running time of the verifier is bounded by $O(T(n))$, where $T(n)$ is the time required to test for heaviness. We give heaviness testers that can test heaviness of an element in the domain $[1, , n]^d$ in time $O((\log n)^d)$. We also also give approximate PCPs with efficient verifiers for recursive bin packing and multidimensional routing.

1 Introduction

Consider a scenario in which the optimal solution to a very large combinatorial optimization problem is desired by a powerful corporation. The corporation hires an independent contractor to actually find the solution. The corporation then would like to trust that the value of the solution is feasible, but might not care about the structure of the solution itself. In particular they would like to have a quick and simple test that checks if the contractor has a good solution by only inspecting a very small portion of the solution itself. Two hypothetical situations in which this might occur are:

- A major corporation wants to fund an international communications network. Data exists for a long history of broadcasts made over currently used networks, including bandwidth, duration, and integrity of all links attempted. The corporation wants to ensure that the new network is powerful enough to handle one hundred times the existing load.
- The services of a trucking company are needed by an (e-)mail-order company to handle all shipping orders, which involves moving large numbers of of boxes between several locations. The mail-order company wants to ensure that the trucking company has sufficient resources to handle the orders.

In both cases, large amounts of typical data are presented to the consulting company, which determines whether or not the load can be handled. The probabilistically checkable-proof (PCP) techniques (cf. [3, 4, 1]) offer ways of verifying such solutions quickly. In

* This work was partially supported by ONR N00014-97-1-0505, MURI, NSF Career grant CCR-9624552, and an Alfred P. Sloan Research Award.The third author was supported in part by an ASSERT grant.

** Part of this work was done while on sabbatical at IBM Almaden Research Center

these protocols a proof is written down which a verifier can trust by inspecting only a constant number of bits of the proof. The PCP model offers efficient mechanisms for verifying any computation performed in NEXP with an efficient verifier. We note that the verifiers in the PCP results all require $\Omega(n)$ time. Approximate PCPs were introduced in [7] for the case when the input data is very large, and even linear time is prohibitive for the verifier. Fast approximate PCPs allow a verifier to ensure that the answer to the optimization problem is at least *almost* correct. Approximate PCPs running in logarithmic or even constant time have been presented in [7] for several combinatorial problems. For example, a proof can be written in such a way as to convince a constant time verifier that there exists a bin-packing which packs a given set of objects into a small number of bins. Other examples include proofs which show the existence of a large flow, a large matching, or a large cut in a graph to a verifier that runs in sublinear time.

Our Results. We consider approximate PCPs for multidimensional bin packing. In particular, we show how a verifier can be quickly convinced that a set of multidimensional objects can be packed into a small number of bins. We also consider the related problems of recursive bin packing and multidimensional routing. Our results generalize the 1-dimensional bin packing results of [7]. The PCPs are more intricate in higher dimensions; for example, the placements and orientations of the blocks within the bin must be considered more carefully. In the l-dimensional case, the approximate PCP o[7] makes use of a property called *heaviness* of an element in a list, introduced by [6]. Essentially, *heaviness* is defined so that testing if an element is heavy can be done very efficiently (logarithmic) in the size of the list and such that all heavy elements in the list are in monotone increasing order. We generalize this notion to the multidimensional case and give heaviness tests which determine the heaviness of a point $x \in [1, . , n]^d$ in time 0((2 log $n)^d$). Then, given a heaviness tester which runs in time T(n), we show how to construct an approximate PCP for binpacking in which the running time of the verifier is $O(T(n))$.

In [9], multidimensional monotonicity testers are given which pass functions f that are monotone and fail functions f if no way of changing the value off at at most ϵ fraction of the inputs will turn f into a monotone function. The query complexity of their tester is $\tilde{O}(d^2n^2r)$ where f is a function from $[n]^d$ to $[r]$. Our multidimensional heaviness tester can also be used to construct a multidimensional monotonicity tester which runs in time $O(T(n))$. However, more recently Dodis *et.* al. [5] have given monotonicity testers that greatly improve on our running times for dimension greater than 2, and are as efficient as ours for dimension 2. This gives hope that more efficient heaviness testers in higher dimensions can also be found.

2 Preliminaries

Notation. We use the notation x $\in_R S$ to indicate x is chosen uniformly and at random from the set S. The notation $[n]$ indicates the interval $[1, \ldots, n]$.

We define a partial ordering relation $\prec$ over integer lattices such that if x and y are d-tuples then x $\prec$ y if and only if $x_i \leq y_i$ for all $i \in \{$ 1, . . . , d$\}$. Consider a function

$f : \mathcal{D}^d \to \mathcal{R}$, we $\mathcal{D}^d$ is $\in \mathcal{D}^d$ $\prec$ y then if $f(x) \leq f(y)$ *y are* in *monotone order*. f is *monotone* $y \in \mathcal{D}^d$ $\prec y$,

Approximate PCP. Π, [7].

Definition 1. *[7] Let $\Delta(\cdot,\cdot)$ be a distance function. A function f is said to have a* *with distance function Δ if there is a randomized verifier V with oracle access to the words of a proof Π such that for all inputs ϵ, and x of size n, the following holds. Let y be the contents of the output tape, then:*

1. *If $\Delta(y, f(x)) = 0$, there is a proof, Π, such that $\mathcal{V}^{\Pi}$ outputs puss with probability at least 3/4 (over the internal coin tosses of V);*
2. *If $\Delta(y, f(x)) > \epsilon$, for all proofs Π', $\mathcal{V}^{\Pi'}$ outputs fail with probability at least 3/4 (over the internal coin tosses of V); and*
3. *V runs in $O(t(\epsilon, n))$ time.*

$1/\delta)$ $\geq 1 - \delta$. [8]. $\Delta(y, f(x)) = \max\{0, 1 - f(x)/y\}$ *approximate lower bound* Π' V" $(1 - \epsilon)y$ Π $\mathcal{V}^{\Pi}$ n $-\epsilon)n$ [7] v, $-\epsilon)v$. V

Heaviness Testing. *heaviness* f. $o \prec$

We give a simple motivating example of a heaviness test for $d = 1$ from [6]. This one-dimensional problem can be viewed as the problem of testing whether a list $L = (f(1), f(2), \ldots, f(n))$ is mostly sorted. Here we assume that the list contains distinct elements (a similar test covers the nondistinct case). Consider the following for testing whether such a list L is mostly sorted: pick a point $x \in L$ uniformly and at random. Perform a binary search on L for the value x. If the search finds x then we call x *heavy*. It is simple to see that if two points x and y are heavy according to this definition, then they are in correct sorted order (since they are each comparable to their common ancestor in the search tree). The definition of a heaviness property is generalized in this paper. We can call a property a *heaviness property* if it implies that points with that property are in monotone order.

Definition *2. Given a domain* $D = [1, \ldots, n]^d$, *a function* $f : D \to R$ *and a property H, we say that H is* a heaviness property if

1. $\forall x < y$ *H(x)* $\wedge$ *H(y) implies f(x)* $\leq$ *f(y)*
2. *In a monotone list all points have property H*

If a point has a heaviness property H then we say that point is *heavy*. There may be many properties which can be tested of points of a domain which are valid heaviness properties. A challenge of designing heaviness tests is to find properties which can be tested efficiently. A heaviness test is a probabilistic procedure which decides the heaviness property with high probability. If a point is not heavy, it should fail this test with high probability, and if a function is perfectly monotone, then every point should pass. Yet it is possible that a function is not monotone, but a tested point is actually heavy. In this case the test may either pass or fail.

Definition 3. *Let* $\mathcal{D}_{\prec} = [1, \ldots, n]^d$ *be a domain, and let* $f : \mathcal{D} \to$ *'R be a function on D. Let* $S(\cdot, \cdot)$ *be a randomized decision procedure on* $\mathcal{D}$. *Given security parameter* δ, *we will say S is a* heaviness test *for x if*

1. *If for all* $x \prec y$, *f(x)* $\leq$ *f(y) then* $S(x, \delta) =$ Pass
2. *If x is not heavy then* $\Pr(S(x, \delta) = \text{Fail}) > 1 - \delta$

The heaviness tests we consider enforce, among other properties, local multidimensional monotonicity of certain functions computed by the prover. It turns out that multidimensional heaviness testing is more involved that the one dimensional version considered in earlier works, and raises a number of interesting questions.

Our results on testing bin-packing solutions are valid for any heaviness property, and require only a constant number of applications of a heaviness test. We give sample heaviness properties and their corresponding tests in Section 4, yet it is an open question whether heaviness properties with more efficient heaviness tests exist. Such tests would immediately improve the efficiency of our approximate PCP verifier for bin-packing.

Permutation Enforcement. Suppose the values of a function f are given for inputs in $[n]$ in the form of a list $y_1, \ldots, y_n$. Suppose further that the prover would like to convince the verifier that the y_i's are distinct, or at least that there are $(1-\epsilon)n$ distinct y_i's. In [7], the following method is suggested: The prover writes array A of length n. $A(j)$

Runtime Verification of Remotely Executed Code using Probabilistically Checkable Proof Systems *

Tuğkan Batu† Ronitt Rubinfeld‡ Patrick White§

Abstract

In this paper we consider the verification and certification of computations that are done remotely. We investigate the use of probabilistically checkable proof (PCP) systems for efficiently certifying such computations. This model can also be applied to verifying security proofs of software downloads. To make the use of PCPs more practical, a new version of Cook's Theorem is given for the RAM model: that is, we show that a correct computation of a RAM can be encoded as a satisfiable boolean formula. We use this result to show that the implementations of PCPs no longer need to be based on a description of the desired computation in terms of a Turing machine program.

1 Introduction

Remote execution of code is an attractive alternative to local execution, when powerful computing resources are available on a network such as the Internet. Yet the recipient of the result of a remote calculation needs to be able to trust that the answer computed is in fact correct. For example, one may be looking for the member in a combinatorial structure which is minimal under some function, such as the optimal tour of a graph, or might like to know that a certain boolean formula is unsatisfiable. Proving the correctness of a proposed answer to each can be performed by a simple exhaustive search, of all feasible tours in the former case, and all valid truth assignments in the second. Both lists are exponentially longer than the input to the problem, and hence than the length of the answer which is being transferred from the powerful computer. Certainly this type of verification is much too costly to be feasible. We consider the approach of applying probabilistically checkable proof systems (PCPs), which allow interactions between a simple and efficient verifying machine and a computationally powerful prover. PCPs provide a general mechanism by which verifier can, in approximately $\log T$ steps, trust the result of a computation requiring T time steps. This takes care of the exponential blow up in our above examples, reducing the time complexity to essentially that of reading the input. In fact, this model is advantageous in any situation in which the computation to be performed takes longer than reading the input. Thus it is even beneficial to use networked computing resources for solving quadratic time problems. Moreover, we can use this model to check the validity of a result without caring about how it is obtained, for example, verifying a satisfying assignment regardless of the process which produced it.

In this paper we consider a model in which a powerful untrusted machine can convince the user that the result of a computation is correct by applying PCP technology. Note that although conceptually the PCP protocols are quite intricate and their correctness is difficult to prove, the actual algorithms employed are not terribly complex. Nonetheless, one unpleasant complication is that all known constructions of PCPs require a description of the computation

*This work was partially supported by ONR N00014-97-1-0505, MURI, NSF Career grant CCR-9624552, and an Alfred P. Sloan Research Award.

†Department of Computer Science, Cornell University, Ithaca, NY 14853 email: batu@cs.cornell.edu.

‡Department of Computer Science, Cornell University, Ithaca, NY 14853 email: ronitt@cs.cornell.edu. Part of this work was done while on sabbatical at IBM Almaden Research Center.

§Department of Computer Science, Cornell University, Ithaca, NY 14853 email: whitepa@cs.cornell.edu.

to be performed in the form of a Turing machine computation. Cook's Theorem is then applied to transform the computation into a satisfiable CNF formula. We instead show how to generate a CNF formula directly from a RAM (random access machine) description, thereby bypassing the need for a Turing Machine description. Even though it is a well-known fact that the RAM and Turing Machine models are equivalent, using the RAM model in this setting proves to be more practical because of the lack of a compiler from any random access model to Turing Machine model. From this point, the PCP protocol can continue unaltered. As a result we redescribe the PCP theorems in terms of a RAM model, which can quite easily be generated by a compiler from a modem high level language, such as C or ML.

1.1 The Model

We assume a computationally bounded user ("Verifier" or "V") is interacting with a very powerful, untrusted computer ("Prover" or "P") over a remote network. There are two specific models we consider. In the first, the Prover is to execute some program M on input x such that both V and P have access to $< M, x >$. Assume that M's computation time is T and that T is prohibitively large for the verifier. The prover generates a PCP proof to convince the verifier that M was executed correctly. It is the surprising result of [ALM+98] based on a long series of papers [LFKN90, BFL91, BFLS90, FGL+96, AS98] that the proof can be written in such a way that V can trust its correctness after inspecting only a constant number of locations in this proof. We use this machinery directly in our first model, which is depicted in Figure 1, in which the prover computes and writes down the proof. The verifier then determines which bits of the proof it wants to see, and the prover transmits these bits to V over the network. In previous PCP results it is assumed that M is given as a Turing machine. In this paper we show that M may be directly encoded as a RAM.

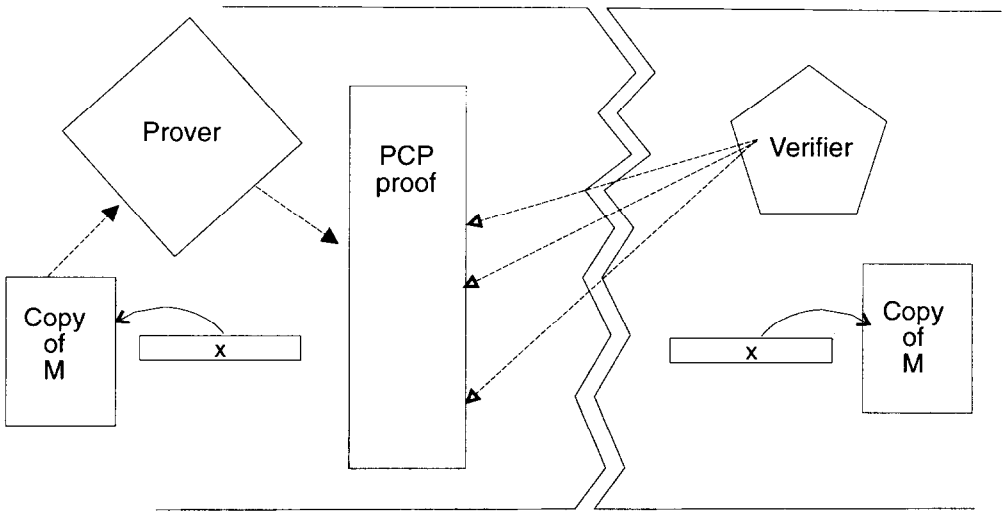


Figure 1: The Basic Model

The second model is an application to secure software download, which has been suggested previously by [DKL+]. V would like to download code (e.g. Netscape) and be convinced that the program has a certain property (e.g. this code will not crash). The first model can be specialized and extended to describe this type of interaction. The Prover provides a **Certificate** of a desired property along with a program. We do not care how the certificate is computed, but we expect that the certificate is so much larger than the code that downloading it is undesirable. The certificate may be very difficult to generate, yet it should be reasonably easy to deterministically certify its correctness. We now assume that both the Prover and Verifier have access to an agreed upon and relatively simple **Certifier,** which takes the program code and the certificate as input and accepts or rejects the certificate in time T.

(The Certifier is here playing the role of M in the first model. The copy of the program is playing the role of x). Without using PCPs, V would have to download the certificate and run the certifier on the certificate. Since the certificate can be large V can instead apply PCP techniques. The verifier spends $O(|x| + \log T)$ steps, where T is the number of steps required to run the certifier. To facilitate this, we again assume the certifier is compiled to RAM code. Using the same techniques as above, P runs the certifier M to check the certificate, and then encodes the run of M as a PCP proof. Now the Verifier acts exactly as in the first model, querying the proof in just a few locations to determine that the certificate is valid. This model is depicted in Figure 2.

As an example of how this model could be used, we will integrate the Proof-Carrying Code (PCC) technique ([NL96]) into our framework. In a PCC system, the code producer, which would be the prover in our model, provides a formal safety proof for a predefined safety policy. This safety proof will act as the certificate in our model. The certifier from our model will be replaced by the proof validator of the PCC system. This proof validator is a reasonably simple program which could be trusted by the code consumer (verifier). At this point, instead of uploading the whole certificate (safety proof) as it would in the PCC system, the code producer will produce and commit to the proof of the certificate being accepted by the certifier as described above. The proof validator is the part of the system that will be encoded as a RAM. The bandwidth of the communication will be reduced drastically as a result.

Proof Commitment One requirement of the PCP protocols is that the proof be written down by the prover and remain unchanged throughout the interaction of the prover and verifier. There are many ways in which the verifier can trust the proof remains unchanged (without downloading the proof). One possibility is to use a trusted third party: P transmits the proof to this third party, and the verifier interacts with the third party assuming that it has no reason to change the proof. Alternatively, one can force P to commit to the proof by using cryptographic techniques of [Mer90], as suggested by [Kil92] (also employed in CS proofs [Mic94] and the work of [DKL+]). This latter scheme introduces only a logarithmic overhead to the running time of the verifier. We assume one of these schemes is employed in what follows.

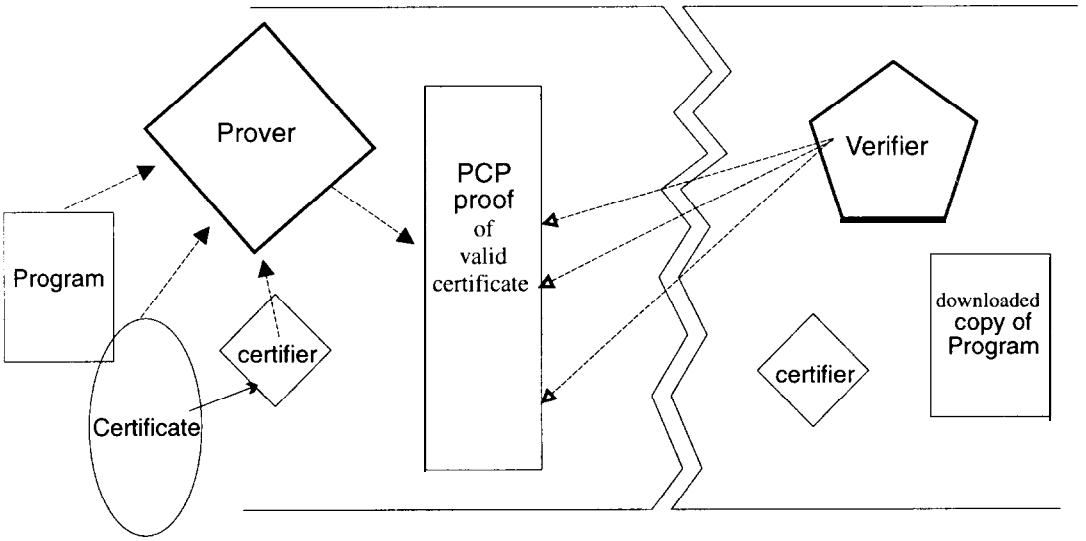


Figure 2: The Verification Model

High level overview of the PCP protocols The various PCP protocols described in the literature share a similar high level outline. To begin, a nondeterministic Turing Machine M is given, along with an input x of length n. The Turing Machine decides x, running in $T(n)$ steps. A prover turns the computation history of M into a 3CNF

formula which is satisfiable if and only if M eventually accepts x. This formula is then transformed into a multivariate polynomial F whose zeros correspond in a one-one fashion to satisfying assignments of the 3CNF formula. T prover finds a value $\vec{x}$ such that $F(\vec{x}) = 0$, and then encodes this value in a special form. The $\vec{x}$ is too long that the verifier does not have time to even look at all of it. However, the encoded version of the proof provid the prover enables the verifier to look at a small number of locations, verify some consistency properties, and verifications pass trust that $F(\vec{x}) = 0$ in $O(n + \log(T))$ time. From the existence of a zero, the verifier concludes that the 3CNF is satisfiable, and thus that M accepted x.

For our purposes, the version of PCP which is most applicable is [BFLS90], which states the verification process in a theorem-proof model. For us the theorem is "This program Q," or "This program returns result x", and the proof is the certificate.

Theorem 1 *(BFLS[1]) A theorem-proof pair < T, P > can be probabilistically verified in time $O(|T| + \log|P|)$.*

Note that the number of locations that the verifier looks at the in proof can be made constant ([PS]), but the total communication overhead is still logarithmic, since the verifier needs to specify the addresses of the locations.

1.2 Our Results

The primary contribution of this paper is a direct reformulation of Cook's Theorem and hence the PCP characterizations of NEXPtime and NP in terms RAMs instead of Turing Machines. Although it is known that RAMs and Turing machines are equivalent in power (cf. [Papa]), we show that a direct application of Cook's Theorem for RAMs is cleanly expressible and easily implemented. This makes it plausible that PCP technology can be employed in an algorithmic setting for realizing computational speed-ups, since it is no longer required that a Turing Machine be constructed to execute the program in question. The [BFLS90] result shows that the machine M can be characterized in terms of Kolmogorov-Uspenskii machines. But again this would require that the prover implement M in terms of a Kolmogorov-Uspenskii machine, or have a compiler from a random access model to the Kolmogorov-Uspenskii model.

Next, we show how this technology can be applied in the setting of result verification. The model we give is quite general and can be applied to the verification of any property which can be computed in nondeterministic exponential time. This question has also been considered in [Mic94, DKL+, FN].

2 Encoding a RAM by a Boolean Formula

2.1 The RAM model

A RAM as described in Papadimitriou [Papa] is a computing device which has direct (i.e. one-step) access to an unlimited number of registers $\{r_1, \ldots r_i, \ldots\}$, each of which can contain an arbitrarily long positive or negative integer. For our purposes, we will assume instead that these registers are space bounded and the largest location used is $S(n)$ which is a parameter of the machine. The input to a RAM is a tape containing a list $\{x_1, x_2, \ldots x_i\}$ of integers, also with random one-step access. The result of a RAM computation is the contents of register 0 after the computation has completed. The RAM program itself is a sequence $(\pi_1, \pi_2, \ldots \pi_m)$ of any of the instructions given in Figure 3. Note that other desirable primitive operations, such as **mult, div, push, pop,** *etc.* can be easily simulated by the given operations, or added directly to the language in a way which will be clear from the expos

2.2 The Encoding

A correct computation of a RAM will be encoded as a satisfiable boolean formula. Each instruction of the program is translated into a boolean sentence which will be quantified over all time steps of the execution o

[1] This version depends on a linear time encoding of the theorem, due to [Spi96].

Fast approximate PCPs *

Funda Ergün† Ravi Kumar‡ Ronitt Rubinfeld§

Abstract

We investigate the problem of when a prover can aid a verifier to reliably compute a function *faster* than if the verifier were to compute the function on its own. We focus on the case when it is enough for the verifier to know that the answer is close to correct. We use a model of proof systems which is based on interactive proof systems, probabilistically checkable proof systems, program checkers, and CS proofs. We develop protocols for several optimization problems, in which the running time of the verifier is significantly less than the size of the input. For example, we give polylogarithmic time protocols for showing the existence of a large cut, a large matching and a small bin packing. In contrast, the protocols used to show that IP = PSPACE, MIP = NEXP and NP = PCP(lgn, 1) [Sha90, BFL91, ALM+98, BFLS90] require a verifier that runs in $\Omega(n)$ time. In the process, we develop a set of tools for use in constructing these proof systems.

1 Introduction

Consider the following scenario: A client sends a computational request to a "consulting" company on the internet, by specifying an input and a computational problem to be solved. The company then computes the answer and sends it back to the client. This scenario is of interest whenever a prover can help a client reliably find the answer to a function faster than the client could compute the function on its own, or whenever the client does not possess the code required to solve the computational problem. An obvious issue that arises, especially in the case that the company does not have a well established reputation, is: why should the client believe the answer to be correct? In this paper, we consider the case when it is enough for the client to know that the answer is close to correct. We show that for a number of interesting functions, a very short dialogue between the client (the verifier) and the company (the prover) can convince the client of the approximate correctness of the answer. In particular, we give examples in which the verifier can be convinced in time sublinear in the input size.

This scenario motivates an investigation of highly efficient proof systems for *specific* computational problems, even when the problems are known to be solvable in polynomial time. Our focus is on finding the best implementations of such proof systems in terms of the running time of the verifier. It is known that IP = PSPACE, MIP = NEXP, NP = PCP($\lg n$, 1) [LFKN90, Sha90, BFL91, ALM+98]. From the work of [BFLS90] and [Spi96], it is possible to construct proof systems for any proof in a reasonable formal system with an $O(n + \lg \ell)$-time verifier, where n is the length of the theorem and ℓ is the length of the proof. Thus we have a good understanding of the set of problems for which it is feasible to find proof systems in which the verifier is efficient and the communication between the prover and verifier is limited. Note that the protocols in the aforementioned results all require that the verifier look at the whole input, and thus do not give sublinear time protocols.

The model we consider, described in Section 2, is based on the models of IP [GMR89], PCP [FRS94], and CS proofs [Mic94], with modifications borrowed from the models of program checking [BK95], approximate program checking [GLR+91], property testing [RS96, GGR98] and spot-checking [EKK+98].

OUR RESULTS. We begin by considering problems that return approximations of optimal solutions for combinatorial optimization problems. We give efficient proof systems for proving good lower bounds on the solution quality to

*This work was partially supported by ONR N00014-97-1-0505, MURI. NSF Career grant CCR-9624552, and an Alfred P. Sloan Research Award.

† Bell Laboratories, 700 Mountain Avenue, Murray Hill, NJ 07974. email: ergun@research.bell-labs.com

‡ IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95 120. email: ravi@almaden.ibm.com

§ Department of Computer Science, Cornell University, Ithaca. NY 14890. email: ronitt@cs . cornell edu. Currently on sabbatical leave at IBM Almaden Research Center.

constraint satisfaction problems, including max cut and max SAT, to a polylogarithmic time verifier. We show how to prove the existence of a near optimal solution of a sparse fractional packing problem to a polylogarithmic time verifier. Our fractional packing protocol can be used for several other problems. For example, we show that it is possible to prove the existence of a large flow, a large matching, or a small bin packing in such a way that the verifier need only spend time nearly linear in the number of vertices (which is sublinear for graphs that are not sparse) in the first case and polylogarithmic time in the latter cases. The size of the proof is nearly linear in the size of the solution to the corresponding search problem and can be computed efficiently by the prover. In all of the above protocols it is also possible to prove the existence of suboptimal solutions, i.e., if the prover knows of a solution of value v, it can prove the existence of a solution of value at least $(1-\epsilon)v$. We next investigate methods of proving additive approximations of bounds on the sizes of unions and intersections of several sets such that the verifier requires only logarithmic time. One application of such protocols is to estimating the size of unions or intersections of several database queries. Finally, we consider spot-checking and property testing and show that more efficient results for testing bipartiteness and element distinctness can be achieved with the aid of a prover.

We develop a new set of tools for use in constructing these proof systems. For example, we give constant time protocols for estimating lower bounds on the size of a set and for estimating lower bounds on sums of n inputs. We develop a constraint enforcement protocol which allows the verifier to ensure that linear upper bound constraints are satisfied without looking at all of the variables involved. Some of our techniques rely on simplifications of the methods of [GS86, For89] by which a prover can convince a verifier that a set is of approximately a certain size.

2 The model

Interactive proof systems (IPS) [GMR89] and probabilistically checkable proof systems (PCPS) [FRS94] (equivalent in power to multiple prover proof systems [BGKW88], see also [FGL+96, AS98, BFLS90]) can be used to convince a polynomial time verifier of the correctness of a decision problem computation. Definitions of IP which parametrize the runtime of the verifier appear in [Con91, FL93]. CS proofs [Mic94] extend the model in several ways: by applying more generally to function computations as well as problems above NEXP; and by restricting the runtime of the prover.

Program result checking [BK95] and self-testing/correcting techniques [BLR93, Lip91] were introduced so that a client could ensure the correctness of a solution to a computation. Program result checkers can be viewed as a special type of proof system for function computations, in which the prover is restricted to answering other instances of the same computational problem. It turns out that all result checkers as well as result checkers in the library setting [BLR93] satisfy the requirements of the model used here.

Proving that results are approximately correct is also related to approximate checking [GLR+91], property testing [RS96, GGR98], and spot-checking [EKK+98], where the goal is to determine whether an answer is close to correct for various notions of closeness. All approximate checkers satisfy the requirement of the model here. Conversely, all of our results can be viewed as spot-checkers which use the aid of a prover.

The model we use is based on the above models and in particular: applies to function computations and decision, optimization, approximation, and search problems; allows the prover to prove the weaker assertion that a solution is approximately correct; parametrizes the run time of the verifier, for example to distinguish linear time verifiers from quadratic time verifiers; and analyzes the run time of the verifier implemented as a RAM machine in order to understand the exact asymptotic complexity of the verifier. We will not assume any bounds on the computation time of the prover.

Both the prover and verifier are interactive RAM machines that have read access to an input and an output tape, read/write access to communication tapes, read access to a public or private source of random bits, and read/write access to private computation tapes. We assume that the verifier can access any word in any tape in constant time.

We give definitions for both the approximate IP and PCP models at the same time. The only distinction between the two is that in the approximate PCP model, the prover is restricted to a function determined before the start of the interaction.

Definition 1 *Let $\Delta(\cdot,\cdot)$ be a distance function. A function f is said to* have *an $t(\epsilon, n)$*-approximate interactive proof (probabilistically checkable proof) system *with distance function* A *if there is a randomized verifier V such that for all inputs ϵ and x of size n, the following holds. Let y be the contents of the output tape, then:*

1. *If $\Delta(y, f(z)) = 0$, there is a (function-restricted) prover $\mathcal{P}$, such that V outputs puss with probability at least 3/4 (over the internal coin tosses of V);*
2. *If* A(y, $f(x)$) > ϵ, *for all* (function-restricted)provers $\mathcal{P}'$, *V outputsfail withprobabilityut least 3/4 (over the internal coin tosses of V); and*
3. *V runs in $O(t(\epsilon, n))$ time.*

The interactive (probabilistically checkable) proof protocol can be repeated $O(\lg 1/\delta)$ times to get confidence $\geq 1-\delta$. We omit all dependence on δ from our protocols throughout this paper.

The choice of the distance function A is problem specific, and determines the ability to construct a proof system, as well as determining how interesting the proof system is.

The usual definitions of interactive proof systems for decision problems require that when y = f(z), an honest prover can convince the verifier of that fact, and when $y \neq f(x)$, no prover can convince the verifier of that. In our model, this is achieved by choosing $\Delta(\cdot,\cdot)$ such that $\Delta(y, y') > \epsilon$ whenever $y \neq y'$ and $\Delta(y, y) = 0$. Note that the output of V is not specified when $0 < \Delta(y, f(x)) \leq \epsilon$. To give a proof system which passes only when $y > (1-\epsilon)f(x)$, one could use the function $A(y, y') = \max\{0, 1 - y/y'\}$. We refer to an IPS/PCPS with this distance function as an *approximate lower bound* IPS/PCPS.

For notational convenience, we often mix notions of interactive and probabilistically checkable proofs by using both within the same protocol, referring to a prover sending information as well as permanently writing down information before the start of the protocol (which corresponds to committing to a set of responses to queries that will be made later in the protocol). These systems can clearly be simulated by a function-restricted prover, since $\mathcal{P}$ can decide on all of its responses before the start of the protocol. Unless otherwise noted, all protocols are described for the PCP model.

INTERACTIVE SPOT-CHECKING MODEL. We give a more general definition of IPS/PCPS which applies to distance functions that correspond to property testing and spot-checking. We define an *interactive-spot-checker*, which is essentially a spot-checker [EKK+98] that is allowed the assistance of a prover.

Definition 2 *Let $\Delta(\cdot,\cdot)$ be a distance function. We say that* V *is an* $t(\epsilon, n)$-interactive-spot-checker (*ISC*) *for f with distance function* A *if, given any input x, claim y for the value of $f(x)$, and ϵ,*

1. *If $\Delta(\langle x, y\rangle, \langle x, f(x)\rangle) = 0$, then there is a function-restricted prover $\mathcal{P}$, such that $\mathcal{V}$ outputspuss with probability at least 3/4 (over the internal coin tosses of V);*
2. *If for all inputs x', $\Delta(\langle x, y\rangle, \langle x', f(x')\rangle) > \epsilon$, then for all function-restricted provers $\mathcal{P}'$, V outputs fail with probability at least 3/4 (over the internal coin tosses of V); and*
3. *V runs in $O(t(\epsilon, n))$ time.*

The choice of the distance function A is problem specific, and determines the ability to spot-check. The condition on the runtime of the spot-checker enforces the "little-oh" property of [BK95], i.e., as long as f depends on all bits of the input, the condition on theruntime of the spot-checker forces the spot-checker to run faster than any correct algorithm for f, which in turn forces the spot-checker to be different than any algorithm for f. Note also that A can be set to 0 for many of the inputs and A need not be computable by the verifier, so that this definition allows interactive proofs for promise problems.

USING PCPS OVER THE INTERNET. When interacting over the internet, the verifier may want some assurance that P is function-restricted, without resorting to having the prover transmit the whole proof in advance of the verification process. One possibility is to use a trusted third party: $\mathcal{P}$ transmits the proof to the third party, and the verifier interacts with the third party assuming that it has no reason to change pieces of the proof. Alternatively, one can bound the running time of $\mathcal{P}$, as is done in the model of CS proofs [Mic94]. Then it is possible to force the prover to commit to the proof in such a way that only provers that are computationally more powerful than the allowed bound are able to change the proof in a convincing way. One can use commitment methods [Mer90] in this setting [Kil92, Mic94, Kil94]. Recently, a third method was proposed by [CMS99] who show that techniques from *private information retrieval* (PIR) [CGKS95] can be used to force a prover to commit to a proof. In this latter work, the essential idea is that since the prover does not know which bit of the proof is being revealed to the verifier, the prover cannot change the proof in a convincing manner. Note that PIR schemes withcomputationally bounded provers are more efficient (cf. [CGKS95, KO97, CMS99]).

RELATED MODELS. Several other works have looked at IPS/PCPS with resource limited verifiers, especially verifiers using logarithmic space. In [Con91, FL93, DS92, FS88], the question of classifying the languages that have interactive proofs with various models of space-bounded verifiers is studied. The work of [DS92, Kil] consider the issue of when zero-knowledge interactive proof systems exist for systems with space bounded verifiers. Related work of [CLSY90] considers the problem of designing untamperable benchmarks for other computers to follow. Their model considers the scenario of a resource-limited computer, which would like to ensure that a (very fast) computer has correctly computed benchmarks without taking any shortcuts. The main difference from this work is that in our model the verifier does not care how the prover computed the answer, only that the answer is correct.

NOTATION. We use $x \in_R S$ to denote that x is chosen uniformly at random from S. We assume the size of a word is b bits and we assume all integer variables fit in a word.

3 Some basic building blocks

3.1 Permutation enforcement (multiset equality)

Given an input list $X = (x_1, \ldots, x_n)$, many of our protocols require that the prover rewrite the list in a different order $Y = \langle y_1, \ldots, y_n\rangle$ (for example, the sorted order). We would like the verifier to be able to ensure that $|X \cap Y| \geq (1-\epsilon)n$. In particular, the verifier should be able to access elements from Y while ensuring that each accessed element corresponds to a single location in X. The difficulty comes from the possibility that neither list is necessarily distinct. One would like to prevent the possibility that an x_i from X was

duplicated more than once in Y, or that two equivalent elements $x_i = x_j$ in X are replaced by only one element in Y. Without the aid of a prover, V requires $\tilde{\Theta}(\sqrt{n})$ time to ensure that $|X \text{ II } Y| \geq (1-\epsilon)n$ [EKK+98]. Here we show that it can be done in $0(1/\epsilon)$ time.

The permutation enforcer consists of two arrays T_1, T_2 of length n, where the contents of location i in T_1 contains a pointer to the location of x_i in Y. Similarly, the contents of location i in T_2 contains a pointer to the location of y_i in X.

Let i be *good* if $T_1[T_2[i]] = i$ and $x_i = y_{T_1[i]}$. Then it is easy to see that:

Lemma 3 $|X \cap \mathbf{Y}| \geq |\{i \mid i \text{ is good}\}|$.

Thus, to verify that $|X \cap Y| \geq (1-\epsilon)n$, the verifier should choose $0(1/\epsilon)$ random i's and output fail if it ever finds an i that is not good. If X = Y, the permutation enforcer will always cause V to pass, and if $|X \cap Y| < (1-\epsilon)n$, no matter what **P** writes in place of the permutation enforcer, V will fail with probability at least $3/4$.

Let $f(X, Y) = 1$ if X = Y and 0 otherwise. Given two multisets X, Y, let $\rho(X, Y)$ be the minimum number of elements that need to be inserted to or deleted from X in order to obtain Y. Then A(((X, Y), $Z\rangle$, ((X', Y'), $f(X', Y')$)) is infinite if either $X \neq X'$ or $Z \neq f(X', Y')$, and otherwise is $\rho(Y, Y')/|Y|$. 0 ne can see that this definition of A ensures V passes only multisets X and Y that are at least close to equal.

Theorem *4 Given two multisets of size n and constant ϵ, there is an $(1/\epsilon)$-ISC for multiset equality with distance function* **A.**

3.2 Element distinctness

Given an input list X = (xi, , $x_n\rangle$, it is often useful for the verifier to ensure that the x_i's are distinct. Here we give a $0(1/\epsilon)$ time protocol by which the verifier can ensure that the number of distinct elements in X is at least $(1-\epsilon)n$. Without the aid of the prover, V requires $\Omega(\sqrt{n})$ time to determine the same [EKK+98]. The protocol we use is a simplification of a protocol given by [For89] in order to allow a prover to convince a verifier of an upper bound on the size of a set. Interestingly, we use the same technique to give a lower bound on the size of a set.

```
Repeat O(1/ε) times:
  V chooses i ∈R [1...n]
  V sends x_i to P
  P returns j to V
  V fails if i ≠ j
```

If X is distinct, then **P** can answer so that V always passes. If the number of distinct elements in X is less than $(1-\epsilon)n$, then for all provers **P'**, V fails with probability at least $3/4$. More formally, let j(X) = 1 if X is distinct and 0 otherwise. Define $\Delta(\langle X, Y), (X', f(X'))\rangle)$ to be infinite if $Y \neq f(X')$, and $\rho(X, X')/|X|$ otherwise (ρ is as defined previously). Note that it is important for the correctness of the protocol that **P** is restricted to a function determined before the start of the interaction.

Theorem 5 *Given a multiset of size n and constant ϵ, there is an $(1/\epsilon)$-ISC for element distinctness with distance function* **A.**

Proof If the multiset X is distinct, **P** can always find $j = i$. If the number of distinct elements in X is less than $(1-\epsilon)n$, the probability that V chooses an i corresponding to a nondistinct element is at least ϵ, and if x_i is not distinct, the probability that $j = i$ is at most $1/2$. Thus, there is a constant c such that after C/E trials, V will fail with probability at least $3/4$. □

A SPACE EFFICIENT PROOF. If the function-restricted **P** in the previous protocol is implemented by having **P** write down the answers to all queries of **P** in advance of the conversation, **P** writes a table of size proportional to a bound on the maximum value of x_i. It is possible to save space, by using an algorithm in which V runs in $0((1/\epsilon) \lg n)$ time: **P** writes a list of ordered pairs containing each input element and its location in the input list (x_i, j) in order sorted by the value of x_i. V then performs binary search to find (x_i, j) and checks $j = i$.

3.3 Proving lower bounds on the size of a set

Given a set S represented by a list enumerating its elements, it is nontrivial to deduce the size of S from the size of the list, since it is not known whether the elements in the list are distinct. Given a method by which V can determine whether a b-bit element x is in S (for example, if S is in fact represented by a list, V could be convinced in constant time that $x \in S$ if **P** sends V a pointer to the location of x in S), V could estimate $|S|/2^b$ to within a multiplicative error of ϵ by sampling. This requires $\Omega(2^b/(\epsilon|S|))$ samples [DKLR95, CEG95]. The two methods described here are more efficient, where the running times are described in terms of γ, an upper bound on a IP (or a PCP) protocol by which **P** can convince V that $x \in S$. The first protocol is faster, but the second protocol (due to [GS86]) can be performed directly in an IP setting.

The following protocols allow **P** to convince V that the size of S is at least $(1-\epsilon)|S|$ for any $\epsilon > 0$. In particular, let p be **P**'s claimed size of S, then if $|S| > p$ the protocol always passes and if $|S| \leq (1-\epsilon)p$ the protocol fails with probability at least $3/4$.

FASTLOWERBOUNDSONSETSIZES IN THE PCP MODEL. The following protocol uses the protocols of the previous sections such that each has probability of error at most 1/8. An auxiliary array A will be used to refer to both an array used to represent the set and the multiset which is defined by its contents.

Spot-Checkers*

Funda Ergün† Sampath Kannan† S Ravi Kumar‡
Ronitt Rubinfeld§ Mahesh Viswanathan†

Abstract

On Labor Day Weekend, the highway patrol sets upspot-checks at random points on the freeways with the intention of deterring a large fraction of motorists from driving incorrectly. We explore a very similar idea in the context of program checking to ascertain with minimal overhead that a program output is *reasonably* correct. Our model of *spot-checking* requires that the spot-checker must run asymptotically much faster than the combined length of the input and output. We then show that the spot-checking model can be applied to problems in a wide range of areas, including problems regarding graphs, sets, and algebra. In particular, we present spot-checkers for sorting, element distinctness, set containment, set equality, total orders, and correctness of group operations. All of our spot-checkers are very simple to state and rely on testing that the input and/or output have certain simple properties that depend on very few bits.

Our sorting spot-checker runs in O(log n) time to check the correctness of the output produced by a sorting algorithm on an input consisting of n numbers. We also show that there is an O(1) spot-checker to check a program that determines whether a given relation is close to a total order. We present a technique for testing in almost linear time whether a given operation is *close to* an associative cancellative operation. In this extended abstract we show the checker under the assumption that the input operation is cancellative and leave the general case for the full version of the paper. In contrast, [RaS96] show that quadratic time is necessary and sufficient to test that a given cancellative operation is associative. This method yields a very efficient tester (over small domains) for all functions satisfying associative functional equations [Acz66]. We also extend this result to test in almost linear time whether the given operation is close to a group operation.

*This work was supported by ONR N00014-97-1-0505, MURI. The second author is also supported by NSF Grant CCR96-199 IO. The third author is also supported by DARPA/AF F30602-95-1-0047. The fourth author is also supported by the NSF Career grant CCR-9624552 and Alfred P. Sloan Research Award. The fifth author is also supported by ARO DAAH04-95-1-0092.

†Email: {fergun@saul, kannan@central, maheshv@gradient}.cis.upenn.edu. Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

‡Email: ravi@almaden ibm corn. IBM Almaden Research Center, San Jose, CA 95 120.

§Email: ronitt@cs.cornell.edu. Department of Computer Science, Cornell University, Ithaca, NY 14853.

1 Introduction

Ensuring the correctness of computer programs is an important yet difficult task. Program result checking[BK89] and self-testing/correcting programs [BLR93, Lip9 1] make runtime checks to certify that the program is giving the right answer. Though efficient, these methods often add small multiplicative factors to the runtime of the programs. Efforts to minimize the overhead due to program checking have been somewhat successful[BW94, Rub94, BGR96] for linear functions. Can this overhead be minimized further by settling for a weaker, yet nontrivial, guarantee on the correctness of the program's output? For example, it could be very useful to know that the program's output is reasonably correct (say, close in Hamming distance to the correct output). Alternatively, for programs that verify whether an input has a particular property, it may be useful to know whether the input is at least close to some input which has the property.

In this paper, we introduce the model of *spot-checking*, which performs only a small amount (sublinear) of additional work in order to check the program's answer. In this context, three prototypical scenarios arise, each of which is captured by our model. In the following, let f be a function purportedly computed by program P that is being spot-checked, and x be an input to f.

- *Functions with small output.* If the output size of the program is smaller than the input size, say $|f(x)| = o(|x|)$ (as is the case for example for decision problems), the spot-checker may read the whole output and

only a small part of the input.

- *Functions with large output.* If the output size of the program is much bigger than the input size, say $|x| = o(|f(x)|)$ (for example, on input a domain D, outputting the table of a binary operation over $D \times D$), the spot-checker may read the whole input but only a small part of the output.
- *Functions for which the input and output are comparable.* If the output size and the input size are about the same order of magnitude, say $|x| = \Theta(|f(x)|)$ (for example, sorting), the spot-checker may only read part of the input and part of the output.

One naive way to define a weaker checker is to ask that whenever the program outputs an incorrect answer, the checker should detect the error with some probability. This definition is disconcerting because it does not preclude the case when the output of the program is very wrong, yet is passed by the checker most of the time. In contrast, our spot-checkers satisfy a very strong condition: if the output of the program is far from being correct, our spot-checkers output FAIL with high probability. More formally:

Definition 1 *Let $A(\cdot,\cdot)$ be a distance function. We say that* C *is an* ϵ-spot-checker *for* f *with distance function* A if

1. *Given any input x and program P (purporting to compute f), and ϵ, C outputs with probability at least 3/4 (over the internal coin tosses of* C) PASS *if* A($\langle x, P(x)\rangle$, $\langle x, f(z)\rangle$) *= 0 and* FAIL *if for all inputs* y, $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) > \epsilon$.
2. *The runtime of* C *is* $o(|x| + |f(x)|)$

The spot-checker can be repeated $O(\lg 1/\delta)$ times to get confidence $1 - \delta$. The choice of the distance function A is problem specific, and determines the ability to spot-check. For example, for programs with small output, one might choose a distance function for which the distance is infinite whenever $P(x) \neq f(y)$, whereas for programs with large output it may be natural to choose a distance function for which the distance is infinite whenever $x \neq y$.

OUR RESULTS. We show that the spot-checking model can be applied to problems in a wide range of areas, including problems regarding graphs, sets, and algebra. We present spot-checkers for sorting, element distinctness, set containment, set equality, total orders, and group operations. All of our spot-checker algorithms are very simple to state and rely on testing that the input and/or output have certain simple properties that depend on very few bits; the non-triviality lies in the choice of the distribution underlying the test. Some of our spot-checkers run much faster than $o(|x| + |f(x)|)$ — for example, our sorting spot-checker runs in $O(\lg |x|)$ time. All of our spot-checkers have the additional property that if the output is incorrect even on one bit, the spot-checker will detect this with a small probability. In order to construct these spot-checkers, we develop several new tools, which we hope will prove useful for constructing spot-checkers for a number of other problems.

One of the techniques that we developed for testing group operations allows us to efficiently test that an operation is associative. Recently in a surprising and elegant result, [RaS96] show how to test that operation $\circ$ is associative in $O(|D|^2)$ steps, rather than the straightforward $O(|D|^3)$. They also show that $\Omega(|D|^2)$ steps are necessary, even for cancellative operations. In contrast, we show how to test that $\circ$ is *close* (equal on most inputs) to some cancellative associative operation $\circ'$ over domain D in $\tilde{O}(|D|)$ steps[1]. We also show how to modify the test to accommodate operations that are not known to be cancellative, in which case the running time increases to $\tilde{O}(n^{3/2})$. Though our test yields a weaker conclusion, we also give a self-corrector for the operation $\circ'$, i.e., a method of computing $\circ'$ correctly for *all* inputs in constant time. This method yields a reasonably efficient tester (over small domains) for all functions satisfying associative functional equations [Acz66].

RELATIONSHIP TO PROPERTY TESTING. A number of interesting result checkers for various problems have been developed (cf., [BK89, BLR93, EKS97, KS96, AHK95, Kan90, BEG+91, ABC+93]). Many of the checkers for numerical problems have used forms *of property testing* (albeit under various names) to ensure that the program's output satisfies certain properties characterizing the function that the program is supposed to compute. For example, efficient property tests that ensure that the program is computing a linear function have been used to construct checkers. In [GGR96], the idea of using property testing *directly on the input* is first proposed. This idea extended the scope of property testing beyond numeric properties. In [GGR96, GR97], property testing is applied to graph problems such as bipartiteness and clique number. The ideas in this paper are inspired by their work.

For the purposes of this exposition, we give a simplified definition of property testing that captures the common features of the definitions given by [RS96, Rub94, GGR96]. Given a domain H and a distribution $\mathcal{D}$ over H, a function f is *e-close* to a function g over $\mathcal{D}$ if $\Pr_{x\in\mathcal{D}}[f(x) \neq g(x)] \leq \epsilon$. A is a *property tester* for a class of functions 3 if for any given ϵ and function f, with high probability (over the coin tosses of A) $\mathcal{A}$ outputs PASS if $f \in \mathcal{F}$ and FAIL if there is no $g \in \mathcal{F}$ such that g and f are ϵ-close.[2]

Our focus on the checking of program results motivates a definition of spot-checkers that is natural for testing input/output relations for a wide range of problems. All previous property testers used a "Hamming-like" distance func-

[1] The notation $\tilde{O}(n)$ suppresses polylogarithmic factors of n.

[2] In fact, the definition of property testing given by [GGR96] is much more general. For example, it allows one to separately consider two different models of the tester's access to f. The first case is when the tester may make queries to f on any input. The second case is when the tester cannot make queries to f but is given a random sequence of $\langle x, f(x)\rangle$ pairs where x is chosen according to $\mathcal{D}$. In our setting, the former is the natural model.

tion. Our general definition of a distance function allows us to construct spot-checkers for set and list problems such as sorting and element distinctness, where the Hamming distance is not useful. In fact, with a proper distance function, all property testers in [GGR96] can be transformed into spot-checkers. One must, however, be careful in choosing the distance function. For instance, consider a program which decides whether an input graph is bipartite or not. Every graph is close to a graph that is not bipartite (just add a triangle), so property testing for nonbipartiteness is trivial. Thus, unless the distance function satisfies a property such as A((x, y), (x, y')) is greater than ϵ when y $\neq y'$, the spot-checker will have an uninteresting behavior.

2 Set and List Problems

2.1 Sorting

Given an input to and output from a sorting program, we show how to determine whether the output of the program is close in edit-distance to the correct sorting of the input, where the edit-distance $\rho(u, v)$ is the number of insertions and deletions required to change string IL into v. The distance function that we use in defining our spot-checker is as follows: for all x, y lists of elements, $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle)$ is infinite if either $x \neq$ y or $|P(x)| \neq$ If(y)] and otherwise is equal to $\rho(P(x), f(y))/|P(x)|$. Since sorting has the property that for all x, $|x| = |f(x)|$, we assume that the program P satisfies $\forall x,\ |x| = |P(x)|$.t is straightforward to extend our techniques to obtain similar results when this is not the case. We also assume that all the elements in our unsorted list are distinct. (This assumption is not necessary for testing for the existence of a long increasing subsequence.)

In Section 2.1.2, we show that the running time of our sorting spot-checker is tight.

2.1.1 The Test

Our 2ϵ-spot-checker first checks if there is a long increasing subsequence in $P(x)$ (Theorem 2). It then checks that the sets *P(x)* and x have a large overlap (Lemma 8). If $P(x)$ and x have an overlap of size at least $(1-\epsilon)n$, where n = $|x|$, and *P(x)* has an increasing subsequence of length at least $(1-\epsilon)n$, then $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) \leq 2\epsilon$.

For $m = O((1/c) \lg 1/\delta)$ and n = O(lg $1/\delta$), the algorithm presented in the figure checks if an input sequence A has a long increasing subsequence by picking random pairs of indices $i < j$ and checking that $A[i] < A[j]$. An obvious way of picking i and j is to pick i uniformly and then pick j to be $i + 1$. Another way is to pick i and j uniformly, making sure that $i <$ j. One can find sequences, however, that pass these tests even though they do not contain long increasing subsequences. The choice of distribution on the pairs i, j is crucial to the correctness of the checker.

```
Procedure Sort-Check(c, δ)
repeat m times
  choose i ∈_R [1, n]
  for k ← 0 ... lg i do
    repeat n times
      choose j ∈_R [1, 2^k]
      if (A[i − j] > A[i]) then return FAIL
  for k ← 0.. . lg(n − i) do
    repeat n times
      choose j ∈_R [1, 2^k]
      if (A[i] > A[i + j] ) then return FAIL
return PASS
```

Theorem 2 *Procedure* Sort-Check(c, δ) *runs in 0(* (l/c) lg *n* lg^2 $1/\delta$) *time, and satisfies:*

- *If A issorted,* Sort-Check(c, δ) = PASS.
- *If A does not have an increasing subsequence of length at least* $(1-c)n$, *then with probability at least* $1-\delta$, Sort-Check(c, δ) =FAIL.

To prove this theorem we need some basic definitions and lemmas.

Definition 3 *The graph* induced *by an array A, of integers having n elements, is the directed graph* G_A, *where* $V(G_A) = \{v_1, \ldots, v_n\}$ *and* $E(G_A) = \{\langle v_i, v_j\rangle \mid i < j$ *and* $A[i] < A[j]\}$.

We now make some trivial observations about such graphs.

Observation 4 *The graph* G_A *induced by an array* $A = \{v_1, v_2, \ldots, v_n\}$ *is transitive, i.e., if* $\langle u, v\rangle \in E(G_A)$ *and* $\langle v, w\rangle \in E(G_A)$ *then* $\langle u, w\rangle \in E(G_A)$.

We shall use the following notation to define neighborhoods of a vertex in some interval.

NOTATION. $\Gamma^+_{(t,t')}(i)$ denotes the set of vertices in the open interval between t and t' that have an incoming edge from v_i. Similarly, $\Gamma^-_{(t,t')}(i)$ denotes the set of vertices between ♦ and ♦ that have an outgoing edge to v_i.

It is useful to define the notion of a *heavy* vertex in such a graph to be one whose in-degree and out-degree, in every 2" interval around it, is a significant fraction of the maximum possible in-degree and out-degree, in that interval.

Definition *5 A vertex* v_i *in the graph* G_A *is said to be* heavy *if for all k,* $0 \leq k \leq \lg i$, $|\Gamma^-_{(i-2^k,i)}(i)| \geq \eta 2^k$ *and for all k,* $0 \leq k \leq \lg(n - i)$, $|\Gamma^+_{(i,i+2^k)}(i)| \geq \eta 2^k$, *where* $\eta = 3/4$.

Theorem 6 *A graph* G_A *induced by an array A, that has* $(1-c)n$ *heavy vertices, has a path of length at least* (1 − c)n.

The theorem follows as a trivial consequence of the following:

Lemma 7 *If v_i and v_j ($i < j$) are heavy vertices in the graph G_A, then $\langle v_i, v_j \rangle \in E(G_A)$.*

Proof Since G_A is transitive, in order to prove the above lemma, all we need to show is that between any two heavy vertices, there is a vertex v_k such that $\langle v_i, v_k \rangle \in E(G_A)$ and $\langle v_k, v_j \rangle \in E(G_A)$.

Let m be such that $2^m \leq (j - i)$, but $2^{(m+1)} \geq (j - i)$. Let $l = (j - i) - 2^m$. Let I be the closed interval $[j - 2^m, i + 2^m]$ with $|I| = (i + 2^m) - (j - 2^m) + 1 = 2^m - l + 1$. Since v_i is a heavy vertex, the number of vertices in I that have an edge from v_i is at least $\eta 2^m - (\{(j - 2^m) - i\}) = \eta 2^m - l$. Similarly, the number of vertices in I, that are adjacent to v_j is at least $\eta 2^m - (\{j - (i + 2^m)\}) = \eta 2^m - 1$.

Now, we use the pigeonhole principle to show that there is a vertex in I that has an incoming edge from i and an outgoing edge to j. By transitivity that there must be an edge from i to j. This is true if $(\eta 2^m - l) + (\eta 2^m - I) \geq (I] = 2^m - 1 + 1$. Since $\eta = 3/4$, this condition holds if $l \leq 2^{m-1}$.

Now consider the case when $l > 2^{m-1}$. In this case we can consider the intervals of size 2^{m+1} to the right of i and to the left of j and apply the same argument based on the pigeonhole principle to complete the proof. ❑

Proof [of Theorem 2] Clearly if the checker returns FAIL, then the array is not sorted.

We will now show that if the induced graph G_A does not have at least $(1 - c)n$ heavy vertices then the checker returns FAIL with probability $1 - \delta$. Assume that G_A has greater that cn light vertices. The checker can fail to detect this if either of the following two cases occurs: (i) the checker only picks heavy vertices, or (ii) the checker fails to detect that a picked vertex is light. A simple application of Chernoff bound shows that the probability of (i) is at most $\delta/2$.

By the definition of a light vertex, say v_i, there is a k such that $|\Gamma^+_{(i,i+2^k)}(i)|$ (or $|\Gamma^-_{(i,i-2^k)}(i)|$) is less than $(3/4)2^k$. The checker looks at every neighborhood; the probability that the checker fails to detect a missing edge when it looks at the k neighborhood (v_j such that $i \leq j \leq i \pm 2^k$) can be shown to be at most $\delta/2$ by an application of Chernoff's bound. Thus the probability of (ii) is at most $\delta/2$. ❑

In order to complete the spot-checker for sorting, we give a method of determining whether two lists A and B (of size n) have a large intersection, where A is presumed to be sorted.

Lemma *8 Given lists A, B of size n, where A is presumed to be sorted. There is a procedure that runs in $O(\lg n)$ time such that if A is sorted and $|A \cap B| = n$, it outputs* PASS *with high probability, and if $|A \cap B| < cn$ for a suitable constant c, it outputs* FAIL *with high probability.*

Remark: The algorithm may also fail if it detects that A is not sorted or is not able to find an element of B in A.

Proof Suppose A is sorted. Then, one can randomly pick $b \in B$ and check if $b \in A$ using binary search. If binary search fails to find b (either because $b \notin A$ or A is wrongly sorted), the test outputs FAIL. Each test takes O(lg n) time, and constant number of tests are sufficient to make the conclusion. ❑

2.1.2 A Lower Bound for Spot-Checking Sorting

We show that any comparison-based spot-checker for sorting running in $o(\lg n)$ time will either fail a completely sorted sequence or pass a sequence that contains no increasing subsequence of length $\Omega(n)$. We do this by describing sets of input sequences that presents a problem for such spot-checkers. We will call these sequences *3-layer-saw-tooth inputs.*

We define k-layer-saw-tooth inputs (k-lst's) inductively. k-lsts take k integer arguments, $(x_1, x_2, . , x_k)$ and are denoted by $lst_k(x_1, x_2, \ldots, x_k)$. $lst_k(x_1, x_2, \ldots, x_k)$ represents the set of sequences in $\mathbb{Z}^{x_1 x_2 \cdots x_k}$ which are comprised of x_k blocks of sequences from $lst_{k-1}(x_1, x_2, . , x_{k-1})$. Moreover, if k is odd, then the largest integer in the i^{th} block is less than the smallest integer in the $(i + 1)^{th}$ block for $1 \leq i < x_k$. If k is even, then the smallest integer in the i^{th} block is greater than the largest integer in the $i(i + 1)^{th}$ block for $1 \leq i < x_k$. Finally to specify these sets of inputs we need to specify the base case. We define lst_1 (xi) to be the set of sequences in $\mathbb{Z}^{x_1}$ which are increasing.

An example $lst_3(3, 3, 2)$ is:

$$\begin{array}{l} \in lst_2(3,3) \\ \sim 1\ 6\ 1\ 7\ 1\ 8\ 1\ 3\ 1\ 4\ 1\ 1\ 5\ 1\ 0\ 1\ 1\ 1\ 2 \\ \in lst_1(3) \end{array}$$

Note that the longest increasing subsequence in $lst_3(i, j, k)$ is of lentgth ik and can be constructed by choosing one $lst_1(i)$ from each $lst_2(i, j)$.

We now show that $o(\lg n)$ comparisons are not enough to spot-check sorting using any comparison-based checker (including that presented in the previous section). Suppose, for contradiction, that there is a checker that runs in f(n) = $\Theta(\lg n/\alpha(n))$ time where $\alpha(n)$ is an unbounded, increasing function of n. Without loss of generality, the checker generates $O(f(n))$ index pairs $(a_1, b_1), . (a_k, b_k)$, where the $a_l < b_l$ for $1 \leq l \leq k$ and returns PASS if and only if, for all l, the value at position a_l is less than the value at position b_l.

Lemma *9 A checker of the kind described above must either* FAIL *a completely sorted sequence or* PASS *a sequence that contains no increasing sequence of length $\Omega(n)$.*

Proof Maintain an array consisting of log n buckets. For each (a_l, a_l) pair generated by the checker, put this pair in the bucket whose index is $\lfloor \lg(b_l - a_l) \rfloor$. It follows that there is a sequence of $c\alpha(n)$ buckets (for some c < 1) such that the probability (over all possible runs of the checker) that

Property Testing of Abelian Group Operations

S Ravi Kumar* Ronitt Rubinfeld†

July 7, 1998

Abstract

Given an n x n table of a cancellative operation o on a domain of size n, we investigate the complexity of determining whether o is close (equal on most pairs of inputs) to an associative, commutative, and cancellative group operation $\circ'$. We show that one can perform such a test in O(n) time. In contrast, quadratic time is necessary and sufficient to test that a given operation is cancellative, associative, and commutative. We give a sub-quadratic algorithm for the case when o is not known to be cancellative. Our techniques for the case when o is not known to be cancellative were later used by [EKK+97] to test that a function o is associative in the same case. Furthermore, we show how to compute $\circ'$ in constant time, given access to $\circ$. We show that our simple test can be used to quickly check the validity of tables of abelian groups and fields. Another application of our results is to testing programs that compute functions which are solutions to certain functional equations.

1 Introduction

Recently in a surprising result, Rajagopalan and Schulman [RaS96] give a randomized algorithm running in $O(n^2)$ time that tests that a binary operation o over a domain of size n is associative; previously the best known algorithm was the straightforward $O(n^3)$ algorithm that tries all triples. They also show that $\Omega(n^2)$ steps are necessary, even for cancellative operations.'

It is easy to see that the complexity of testing whether an operation over a domain of size n is commutative is $\Theta(n^2)$. We show that this lower bound holds even when $\circ$ is known to be cancellative. We also show that $\Omega(n^2)$ steps are required to know if $\circ$ is both associative and commutative, even when o is known to be cancellative.

The main contribution of this paper is to show that the situation is quite different for determining whether a binary operation is close (equal on most pairs of inputs) to being both associative and

'IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. Email: ravi@almaden.ibm.com.
Most of this work was done while the author was at Cornell University, supported by ONR N00014-97-1-0505, MURI and DARPA/AFF30602-95-1-0047.

'Department of Computer Science, Cornell University, Ithaca, NY 14853. Supported by ONR N00014-97-1-0505, MURI, the NSF Career grant CCR-9624552 and the Alfred P. Sloan Research Award. Email: ronitt@cs.cornell.edu.

'Let $T_\circ$ be the n x n Cayley table corresponding to $\circ$. $\circ$ is said to be *cancellative* if $\forall a, b, c \in G$, $(a \circ c = b \circ c) \Rightarrow (a = b)$ and $\forall a, b, c \in G, (a \circ b = a \circ c) \Rightarrow (b = c)$.

commutative. This gives a ***property tester*** as defined in [RS96, GGR96] for the property of being simultaneously associative and commutative. In fact, given a table describing a cancellative binary operation o on a domain G of size n, we give a randomized algorithm that tests in $O(n)$ time if o is close to a cancellative o' that is both commutative and associative. We also give a randomized algorithm that computes o' correctly for *all* inputs in constant time by making queries to the table for $\circ$. We extend these results to the case when o is not necessarily known to be cancellative. In this case, we give a randomized algorithm that runs in $O(n^{3/2})$ time for the same problem. An interesting open question is whether one can test that $\circ$ is close to a cancellative, commutative and associative operation in sublinear time (see Section 7).

Previously, [EKK+97] show how to test that a cancellative operation $\circ$ is close to some cancellative associative operation o' in $O(n \lg^2 n)$ time. They also give a randomized algorithm which computes $\circ'$ correctly for all inputs in constant time. Our techniques are similar to theirs at a high level, but since different equations are used to perform the test, the low level manipulations are necessarily different. In particular, the efficiency of our test is better. Our techniques for the case where o is not known to be cancellative were later used by [EKK+97] in order to give a subquadratic time algorithm for associativity testing in general case.

We then show how to use the associativity-commutativity test, in conjunction with previous results on testing that an operation is a homomorphism [BLR93], to test that the domain G together with cancellative operations o and o "resembles" a field. More precisely, if o (resp. o) is known to be cancellative, we can test in $O(n)$ time if o (resp. o) is close to o' (resp. $\diamond'$) such that (G, o', $\diamond'$) is a field. If $\circ$ (resp. o) is not known to be cancellative, we can test in $O(n^{3/2})$ time if o (resp. o) is close to o' (resp. $\diamond'$) such that (G, o', $\diamond'$) is a field.

Another motivation for studying this problem is its application to program checking, self-testing, and self-correcting [BK89, BLR93, Lip91]. Using techniques from [Rub94], our method yields a reasonably efficient self-tester and self-corrector (over small domains) for all functions that are solutions to the composite functional equation

$$F[F[x,y],z] = F[x,F[y,z]].$$

Such functional equations are used to characterize algebraic structures more general than semigroups [Acz66]. For instance, if S, P are sets and F : S x $P \rightarrow$ S, the related functional equation

$$F[F[x,u],v] = F[F[x,v],u]$$

for all x, F [x, u] , F [x, v] $\in$ S and u, v $\in P$ characterizes an abelian group in the following manner: suppose F [x, u] = y has for all x, y$\in$ S at least one solution $u \in P$. Fix an arbitrary $x_0 \in$ S and define x o y $\stackrel{\Delta}{=} F$ [y, u] where u is a solution of the equation $F[x_0, u]$ = x. Then, it can be shown that (S, o) is an abelian group.

OTHER RELATED WORK. Variants of property testing (under various names) has been defined by [RS96, GGR96]. Property testers have been given for algebraic properties such as linearity and low degree testing (cf. [BLR93, RS96, FGL+91, AS92, ALM+92, BCH+95, Tre98]), graph properties such as bipartiteness, clique number, transitive tournament [GGR96, GR97, GR98, EKK+97], and comparison-based problems such as the monotonicity of a list and of boolean functions [EKK+97, GGLR98].

ORGANIZATION. Section 3 contains the testing algorithm for the cancellative case. Section 4 combines this algorithm with some additional machinery to obtain a tester for fields. An $\Omega(n^2)$ lower bound for commutativity is proved in Section 5. Finally, Section 6 gives a tester for the non-cancellative case.

2 Definitions and Notation

We denote an element a chosen randomly with uniform distribution from G by $a \in_R G$. We will also use the same notation to denote that α is distributed uniformly in G. We use $\alpha \in_D G$ to denote that a is distributed according to distribution *D*. For random variables a, β, γ, we often reason about the probability that $a = \gamma$ by using an intermediate variable β, using $\Pr[\alpha = \gamma] \geq \Pr[\alpha = \beta = \gamma] \geq 1 - \Pr[\alpha \neq \beta] - \Pr[\beta \neq \gamma]$. The notation $\Pr_\alpha[\cdot]$ is synonymous with $\Pr_{\alpha \in_R G}[\cdot]$.

The L_1 *-distance* between two discrete distributions *D, D'* on G is defined to be $\sum_{x \in G} |D(x) - D'(x)|$ where *D(x)* (resp. *D'(x))* denotes the probability of generating x according to *D* (resp. D'). A distribution is ε-*uniform* if its L_1 -distance to the uniform distribution is $\leq \varepsilon$.

Let $T_\circ$ be the n x n cancellative Cayley table corresponding to $\circ$. In this case, each row and column of $T_\circ$ is a permutation of elements in G. Thus, a cancellative $T_\circ$ is isomorphic to an n x n latin square. Using these, we make can the following simple observation.

Observation 1 *If $\circ$ is cancellative, thenforany $b \in G$, if $\alpha \in_R G \Rightarrow \alpha \circ b \in_R G$.*

Note that if o is cancellative then for any *a*, if $\alpha_1 \in_R$ G and α_1 o $\alpha_2 = a$, then $\alpha_2 \in_R$ G, though α_2 is not independent from al. We now define what it means for two operations to be close to each other.

Definition 2 *Let o and o' be binary operations. o is* E-close *to o' if* $\Pr_{\alpha,\beta}[\alpha \circ \beta = \alpha \circ' \beta] \geq 1 - \varepsilon$.

We extend this notion to define an almost abelian group.

Definition 3 *Let o be a closed binary operation on G. $(G, \circ)$ is an* ε-abelian group *if there exists a binary operation o' that is E-close to $\circ$ such that $(G, \circ')$ is an abelian group.*

This notion can be extended to fields as well.

Definition 4 *Let $\circ, o$ be a closed binary operations on G. $(G, \circ, o)$ is an $(\varepsilon_1, \varepsilon_2)$*-field *if there exist binary operations $\circ'$ (resp. $\diamond'$) that is ε_1 -close to o (resp. ε_2-close to* 0*) such that (G, o', $\diamond'$) is a jield.*

REMARK ON CONFIDENCE. Our tests rely on random sampling to estimate various probabilities. Roughly, it requires $O(\frac{1}{\varepsilon} \ln \frac{1}{\rho})$ trials to estimate a quantity of magnitude ε with a confidence of p. This is achievable by standard Chernoff-Hoeffding type analysis [Che52].

3 Associativity-Commutativity Test: The Cancellative Case

In this section we assume that o is cancellative. Later, in Section 6, we show how to extend these techniques to the non-cancellative case.

We show that testing whether a cancellative o is E-close to a cancellative, associative, and commutative $\circ'$ can be done in randomized *O(n)* time (Section 3.1). We also show that there is an $\Omega(n^2)$ lower bound to check if $\circ$ is fully associative and commutative (Section 3.2). We then use the testing algorithm to test if (G, o) is an ε-abelian group (Section 3.3).

3.1 The Upper Bound

We use the following equation (which we call the *AC-property)* to test:

$$(a o b) o c = a o (c o b).$$

We prove the following theorem which shows that if a cancellative o satisfies some conditions that can be tested in *O(n)* time, then it is 5ε-close to a cancellative, associative, and commutative $\circ'$. Furthermore, this theorem also shows that $\circ'$ can be computed (by making queries to $\circ$) in constant time with high probability of correctness, assuming there is an oracle that given $a, \alpha_1 \in G$ returns an $\alpha_2 \in G$ such that $\alpha_1 \circ \alpha_2 = a$. Even if such an oracle is not available, one can precompute for all $b \in G$, $0(1)$ many random β_1, β_2 such that $\beta_1 \circ \beta_2 = b$ in $O(n \lg n)$ time by a coupon-collector method.

Theorem 5 *Let* $\varepsilon < 1/21$. *If o is cancellative and satisfies*

(1) $\Pr_\alpha[\forall b, \alpha \circ b = boa] \geq 1 - \varepsilon$,

(2) $\Pr_{\beta,\gamma}[\forall a, (a \circ \beta) \circ \gamma = a \circ (\gamma \circ \beta)] \geq 1 - \varepsilon$,

(3) $\Pr_{\alpha,\gamma}[\forall b, (a\ o\ b)\ o\ \gamma = \alpha\ o\ (yo\ b)] \geq 1 - E$, *and*

(4) $\Pr_{\alpha,\beta}[\forall c, (\alpha \circ \beta) \circ c = \alpha \circ (c \circ \beta)] \geq 1 - \varepsilon$,

then there is an $\circ'$ *such that*

(1) *0' is cancellative,*

(2) $\forall a, b, c,\ a \circ' (b \circ' c) = (a \circ' b) \circ' c$,

(3) $\forall a, b,\ a \circ' b = bo'a$,

(4) $\circ'$ *is* 5ε*-close to o, and*

(5) *o' is computable in constant time, given access to o.*

Proof Outline: Let maj denote the majority function which returns the element that occurs the most number of times in a (multi)set. Define the following binary operation o' as follows: for $a, b \in G$, define

$$a \circ' b \stackrel{\Delta}{=} \underset{\beta_1 \circ \beta_2 = b}{\mathrm{maj}} \{(a \circ \beta_2) \circ \beta_1\}.$$

An Approximate L^1-Difference Algorithm for Massive Data Streams*

Joan Feigenbaum
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
jf@research.att.com

Sampath Kannan†
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
skannan@research.att.com

Martin Strauss
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
mstrauss@research.att.com

Mahesh Viswanathan‡
Computer and Information Sciences
University of Pennsylvania
Philadelphia, PA 19104 USA
maheshv@saul.cis.upenn.edu

July 27, 1999

Abstract

Massive data sets are increasingly important in a wide range of applications, including observational sciences, product marketing, and monitoring and operations of large systems. In network operations, raw data typically arrive in *streams*,and decisions must be made by algorithms that make one pass over each stream, throw much of the raw data away, and produce "synopses" or "sketches" for further processing. Moreover, network-generated massive data sets are often *distributed:* Several different, physically separated network elements may receive or generate data streams that, together, comprise one logical data set; to be of use in operations, the streams must be analyzed locally and their synopses sent to a central operations facility. The enormous scale, distributed nature, and one-pass processing requirement on the data sets of interest must be addressed with new algorithmic techniques.

We present one fundamental new technique here: a space-efficient, one-pass algorithm for approximating the L^1 difference $\sum_i |a_i - b_i|$ between two functions, when the function values a_i and b_i are given as data streams, and their order is chosen by an adversary. Our main technical innovation, which may be of interest outside the realm of massive data stream algorithmics, is a method of constructing families $\{V_j\}$ of limited-independence random variables that are range-summable, by which we mean that $\sum_{j=0}^{c-1} V_j(s)$ is computable in time $polylog(c)$, for all seeds s. Our L^1-difference algorithm can be viewed as a "sketching" algorithm, in the sense of [Broder, Charikar, Frieze, and Mitzenmacher, STOC 98, pp. 327-336], and our technique performs better than that of Broder *et* al. when used to approximate the symmetric difference of two sets with small symmetric difference.

*This paper is an expanded version of the authors' 1999 IEEE Symp. on Foundations of Computer Science extended abstract with the same title

†On leave from the Univ. of Pennsylvania. Part of this work was done at the Univ. of Pennsylvania, supported by grants NSF CCR96-19910 and ARO DAAH04-95-1-0092.

‡Supported by grant ONR N00014-97-1-0505, MURI.

1 Introduction

Massive data sets are increasingly important in a wide range of applications, including observational sciences, product marketing, and monitoring and operations of large systems. In network operations, raw data typically arrive in streams, and decisions must be made by algorithms that make one pass over each stream, throw much of the raw data away, and produce "synopses" or "sketches" for further processing. Moreover, network-generated massive data sets are often distributed: Several different, physically separated network elements may receive or generate data streams that, together, comprise one logical data set; to be of use in operations, the streams must be analyzed locally and their synopses sent to a central operations facility. The enormous scale, distributed nature, and one-pass processing requirement on the data sets of interest must be addressed with new algorithmic techniques.

We present one fundamental new technique here: a space-efficient, one-pass algorithm for approximating the L^1 difference $\sum_i |a_i - b_i|$ between two functions, when the function values a_i and b_i are given as data streams, and their order is chosen by an adversary. This algorithm fits naturally into a toolkit for Internet-traffic monitoring. For example, Cisco routers can now be instrumented with the NetFlow feature [CN98]. As packets travel through the router, the NetFlow software produces summary statistics on each *flow*.[1] Three of the fields in the flow records are source IP-address, destination IP-address, and total number of bytes of data in the flow. At the end of a day (or a week, or an hour, depending on what the appropriate monitoring interval is and how much local storage is available), the router (or, more accurately, a computer that has been "hooked up" to the router for monitoring purposes) can assemble a set of values $(x, f_t(x))$, where x is a source-destination pair, and $f_t(x)$ is the total number of bytes sent from the source to the destination during a time interval t. The L^1 difference between two such functions assembled during different intervals or at different routers is a good indication of the extent to which traffic patterns differ.

Our algorithm allows the routers and a central control and storage facility to compute L^1 differences efficiently under a variety of constraints. First, a router may want the L^1 difference between f_t and f_{t+1}. The router can store a small "sketch" of f_t, throw out all other information about f_t, and still be able to approximate $\|f_t - f_{t+1}\|_1$ from the sketch of f_t and (a sketch of) f_{t+1}.

The functions $f_t^{(i)}$ assembled at each of several remote routers R_i at time t may be sent to a central tape-storage facility C. As the data are written to tape, C may want to compute the L^1 difference between $f_t^{(1)}$ and $f_t^{(2)}$, but this computation presents several challenges. First, each router R_i should transmit its statistical data when R_i's load is low and the R_i-C paths have extra capacity; therefore, the data may arrive at C from the R_i's in an arbitrarily interleaved manner. Also, typically the x's for which $f(x) \neq 0$ constitute a small fraction of all x's; thus, R_i should only transmit $(x, f_t^{(i)}(x))$ when $f_t^{(i)}(x) \neq 0$. The set of transmitted x's is not predictable by C. Finally, because of the huge size of these streams,[2] the central facility will not want to buffer them in the course of writing them to tape (and cannot read from one part of the tape while writing to another), and telling R_i to pause is not always possible. Nevertheless, our algorithm supports approximating the L^1 difference between $f_t^{(1)}$ and $f_t^{(2)}$ at C, because it requires little workspace, requires little time to process each incoming item, and can process in one pass all the values of both functions $\{(x, f_t^{(1)}(x))\} \cup \{(x, f_t^{(2)}(x))\}$ in any permutation.

[1] Roughly speaking, a "flow" is a semantically coherent sequence of packets sent by the source and reassembled and interpreted at the destination. Any precise definition of "flow" would have to depend on the application(s) that the source and destination processes were using to produce and interpret the packets. From the router's point of view, a flow is just a set of packets with the same source and destination IP-addresses whose arrival times at the routers are close enough, for a tunable definition of "close."

[2] A WorldNet gateway router now generates more that 10Gb of NetFlow data each day.

Our L^1-difference algorithm achieves the following performance:

> Consider two data streams of length at most ***n,*** each representing the non-zero points on the graph of an integer-valued function on a domain of size n. Assume that the maximum value of either function on this domain is M. Then a one-pass streaming algorithm can compute with probability $1 - \epsilon$ an approximation A to the L^1-difference B of the two functions, such that $|A - B| \leq \lambda B$, using space $O(\log(M)$ log(n) $\log(1/\epsilon)/\lambda^2)$ and time O(log(n) log log(n) + log(M) $\log(1/\epsilon)/\lambda^2)$ to process each item. The input streams may be interleaved in an arbitrary (adversarial) order.

The main technical innovation used in this algorithm is a limited-independence random-variable construction that may prove useful in other contexts:

> A family $\{V_j(s)\}$ of uniform ± 1-valued random variables is called range-summable if $\sum_0^{c-1} V_j(s)$ can be computed in time $polylog(c)$, for all seeds s. We construct range-summable families of random variables that are n^2-*bad 4-wise independent.*[3]

The property of n^2-bad 4-wise independence suffices for the time- and space-bounds on our algorithm. Construction of truly 4-wise independent, range-summable random-variable families for which the range sums can be computed as efficiently as in our construction remains open.

The rest of this paper is organized as follows. In Section 2, we give precise statements of our "streaming" model of computation and complexity measures for streaming and sketching algorithms. In Section 3, we present our main technical results. Section 4 explains the relationship of our algorithm to other recent work, including that of Broder *et al.* [BCFM98] on sketching and that of Alon *et al.* [AMS96] on frequency moments.

2 Models of Computation

Our model is closely related to that of Henzinger, Raghavan, and Rajagopalan [HRR98]. We also describe a related sketch model that has been used, e.g., in [BCFM98].

2.1 The Streaming Model

As in [HRR98], a *data stream* is a sequence of data items $\sigma_1, \sigma_2, \ldots, \sigma_n$ such that, on each *puss* through the stream, the items are read once in increasing order of their indices. We assume the items σ_i come from a set of size M, so that each σ_i has size log *M*. In our computational model, we assume that the input is one or more data streams. We focus on two resources-the *workspace* required in bits and the *time* to *process* an item in the stream. An algorithm will typically also require pre- and post-processing time, but usually applications can afford more time for these tasks.

Definition 1 The complexity class PASST($s(\epsilon, \lambda,$ n, *M), $t(\epsilon, \lambda, n, M)$)* (to be read as "probably approximately streaming space complexity $s(\epsilon, \lambda,$ n, *M)* and time complexity $t(\epsilon, \lambda,$ n, *M)*") contains those functions f for which one can output a random variable X such that $|X - f| < \lambda f$ with probability at least $1 - \epsilon$ and computation of X can be done by making a single pass over the data, using workspace at most $s(\epsilon, \lambda,$ ***n,*** *M)* and taking time at most $t(\epsilon, \lambda, n,$ *M)* to process each of the ***n*** items, each of which is in the range 0 to $M - 1$.

If s = ***t,*** we also write PASST(s) for PASST(s, ***t).*** I

[3] The property of n^2-bad 4-wise independence is defined precisely in Section 3 below.

We will also abuse notation and write $A \in \mathrm{PASST}(s, t)$ to indicate that an algorithm A for f witnesses that $f \in \mathrm{PASST}(s, t)$.

2.2 The Sketch Model

Sketches were used in [BCFM98] to check whether two documents are nearly duplicates. A sketch can also be regarded as a synopsis *data structure* [GM98].

Definition 2 The complexity class $\mathrm{PAS}(s(\epsilon, \lambda, n, M))$ (to be read as "probably approximately sketch complexity $s(\epsilon, \lambda, n, M)$") contains those functions f : X x X $\rightarrow Z$ of two inputs for which there exists a set S of size 2^s, a randomized *sketch function* h : X $\rightarrow$ S, and a randomized *reconstruction function* ρ : S x S $\rightarrow Z$ such that, for all $x_1, x_2 \in$ X, with probability at least $1 - \epsilon$, $|\rho(h(x_1), h(x_2)) - f(x_1, x_2)| < \lambda f(x_1, x_2)$. ∎

By "randomized function" of k inputs, we mean a function of k + 1 variables. The first input is distinguished as the source of randomness. It is not necessary that, for all settings of the last k inputs, for most settings of the first input, the function outputs the same value.

Note that we can also define the sketch complexity of a function f : X x Y $\rightarrow Z$ for X $\neq$ Y. There may be two different sketch functions involved.

There are connections between the sketch model and the streaming model. Let XY denote the set of concatenations of $x \in$ X with y $\in$ Y. It has been noted in [KN97] and elsewhere that a function on XY with low streaming complexity also has low one-round communication complexity (regarded as a function on X x Y), because it suffices to communicate the memory contents of the hypothesized streaming algorithm after reading the X part of the input. Sometimes one can also produce a low sketch-complexity algorithm from an algorithm with low streaming complexity.[4] Our main result is an example.

Also, in practice, it may be useful for the sketch function h to have low streaming complexity. If the set X is large enough to warrant sketching, then it may also warrant processing by an efficient streaming algorithm.

Formally, we have:

Theorem *3 If $f \in \mathrm{PAS}(s(\epsilon, \lambda, n, M))$ via sketch function $h \in \mathrm{PASST}(s(\epsilon, \lambda, n, M), t(\epsilon, \lambda, n, M))$, then $f \in \mathrm{PASST}(2s(\epsilon, \lambda, n/2, M), t(\epsilon, \lambda, n/2, M))$.*

2.3 Arithmetic and Bit Complexity

Often one will run a streaming algorithm on a stream of n items of size log M on a computer with word size at least max(log M, log n). We assume that the following operations can be performed in constant time on words:

- Copy x into y
- Shift the bits of x one place to the left or one place to the right.
- Perform the bitwise AND, OR, or XOR of x and y.
- Add x and y or subtract x from y.

[4] This is not always possible, e.g., not if $f(x, y)$ is the x'th bit of y.

Part IV

Complexity

John Mitchell, Mark Mitchell, and Andre Scedrov: "A Linguistic Characterization of Bounded Oracle Computation and Probabilistic Polynomial Time", in the Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science — FOCS'98, pp. 725-733, IEEE Computer Society Press, Palo Alto, CA, November 1998.

Full paper: ftp://www.cis.upenn.edu/pub/papers/scedrov/ppoly.ps.gz

Patrick Lincoln, John Mitchell, Mark Mitchell, and Andre Scedrov: "A Probabilistic PolyTime Framework", in the Proceedings of the 5th ACM Conference on Computer and Communication Security — CCS-5, San Francisco, CA, November 1998

Full paper: http: //www.csl.sri.com/~lincoln/papers/acm-98.ps

Iliano Cervesato, Massimo Franceschet, and Angelo Montanari: "The Complexity of Model Checking in Modal Event Calculi with Quantifiers", *Electronic Transactions in Artificial Intelligence,* vol. 2(1–2), January-June 1998, pp. l-24.

Full paper: http: //www.stanford.edu/~iliano/papers/etai98.ps.gz

Joan Feigenbaum, Sampath Kannan, Moshe Y. Vardi, and Mahesh Viswanathan: "Complexity of Problems on Graphs Represented as OBDDs", in the Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science — STACS'98, Paris, France, February 1998.

Full paper: file://ftp.cis.upenn.edu/pub/papers/kannan/mahesh.ps.Z

Patrick Lincoln, John Mitchell, and *Andre Scedrov:* "Optimization Complexity of Linear Logic Proof Games", accepted for publication in *Theoretical Computer Science.*

Fullpaper: `ftp://www.cis.upenn.edu/pub/papers/scedrov/approxTCS.ps.gz`

A Linguistic Characterization of Bounded Oracle Computation and Probabilistic Polynomial Time

J. Mitchell * M. Mitchell†
Stanford University
{mitchell,mmitchel} @cs.stanford.edu

A. Scedrov ‡
University of Pennsylvania
scedrov@saul.cis.upenn.edu

Abstract

We present a higher-order functional notation for polynomial-time computation with an arbitrary 0, 1-valued oracle. This formulation provides a linguistic characterization for classes such as NP *and* BPP, *as well as a notation for probabilistic polynomial-time functions. The language is derived from Hofmanns adaptation of Bellantoni-Cook safe recursion, extended to oracle computation via work derivedfrom that of Kapron and Cook. Like Hofmann's language, ours is an applied typed lambda calculus with complexity bounds enforced by a type system. The type system uses a modal operator to distinguish between two sorts of numerical expressions. Recursion can take place on only one of these sorts. The proof that the language captures precisely oracle polynomial time is model-theoretic, using adaptations of various techniques from category theory.*

1 Introduction

In 1964, Cobham proposed a characterization of feasible functions that is based on a binary-numeral form of primitive recursion[7]. In Cobham's definition, primitive recursion is restricted in an essentially ad hoc way, by requiring that any function defined by primitive recursion be bounded above by some other function already shown to be computable in polynomial time. Over the past 30+ years, Cobham's recursion scheme has been repeatedly analyzed and reworked. One motivation for this line of research has been to find a "logical" characterization of polynomial time that does not contain any obvious use of clocks or other mechanisms that count the number of computation steps. Another motivation has been to obtain a characterization of higher-order polynomial time[8].

We present a higher-order typed programming language characterizing polynomial-time computation with a 0, l-valued oracle. Since probabilistic computation may be characterized as computation relative to an oracle, where the oracle is chosenprobabilistically, we also view the language as characterizing probabilistic polynomial time computation. The principal complexity-theoretic property of the language is that every function (of a certain syntactic type) that is definable in the language can be computed in time that is bounded by a polynomial function of the input, independent of the oracle. The fact that the running time is bounded by the same polynomial, for all oracles, makes it possible to capture complexity classes such as NP and BPP. For example, NP is the class of languages L for which there exists a particular kind of terms in the language. In particular, these terms must be such that for every candidate word s, there is some oracle ϕ whereby t, computing with the assistance of the oracle ϕ, decides $s \in L$. Because we know that the language captures precisely the oracle polynomial time functions, we do not need to mention polynomial time explicitly in the definition.

For those not familiar with the area, it may be helpful to point out that there are several incomparable but equally compelling definitions of the class of computable functions of higher type (e.g., functions with function inputs). Therefore, we may also expect to find several apparently reasonable classes ofhigher-

*Partially supported byDoD MURI "Semantic Consistency in Information Exchange," ONR GrantN00014-97-1-0505, with additional support from NSF CCR-9629754.

†Additional support from Stanford University Fellowship.

‡ Additional support from NSF Grant CCR-9800785.

order polynomial-time functions. (This general issue is discussed in [8], for example.) One natural approach to higher-order polynomial time is through programming languages that respect resource bounds. More specifically, suppose we can define a language that contains function symbols and such that every natural number function definable in the language can be computed in polynomial time. Then we may obtain a class of "higher-order polynomial time functions" by treating expressions in the language as functions of the higher-order variables they contain.

One complication that arises with time-bounded computation, but not with computability independent of resource bounds, is that computation time may depend on the values of the input function. For example, consider a function $f(x, g)$ with natural number input x and function input g. Suppose that on input x and g, the function f applies some polynomial-time function **h** to g(x). If g is some arbitrary input function, then we have no reason to expect the size of g(x) to be bounded by some polynomial in $|x|$. If g(x) is exponentially larger, for example, then the computation of *h(g(x))* may be exponential, even if we just count the running time of *h* and assume that g(x) is obtained in a single step. For this reason, we restrict the oracle functions to taking only the value 0 and 1. (Of course, ordinary functions definable in the language may take arbitrary values.)

The starting point for the work presented here is a higher-order typed lambda calculus, containing function symbols of arbitrary type and a form of recursion operator called *safe recursion.* This calculus and associated complexity analysis were developed by Hofmann [11], building on work by Bellantoni and Cook [3, 5]. In brief, Bellantoni-Cook safe recursion achieves the same goal as Cobham's restricted form of primitive recursion, but through different means. Instead of an explicit bound, there is implicit control over complexity through the use of two separate lists of arguments. One list of arguments, called the *normal* arguments, may be used in any way. Arguments from the second list, referred to as *safe,* cannot be used as the recursion argument in any nested safe recursion. Through this mechanism, described in more detail in Section 3, it is possible to define all polynomial-time functions, but it is not possible to nest such computations a variable number of times. While Bellantoni and Cook worked in a first-order framework similar to ordinary primitive recursive notation, Hofmann captured the safe/normal distinction through a type system that brings the system closer to a convenient programming language. In particular, Hofmann's framework allows the use of higher-order functions.

Our extension of Hofmann's system retains the typed lambda calculus framework, but allows the use of an oracle function not assumed definable within the language. Since a nondeterministic or probabilistic machine uses a different "choice" or random bit at each branch point, we formulate our oracle primitive as a basic operation that returns the next bit of the oracle sequence each time it is called. This makes it easy to show how a probabilistic or nondetetministic algorithm can be written in our language.

Although it has little direct bearing on the results described here, our motivation for this work is the study of security properties. Specifically, as described in [14], we have developed a language for defining concurrent systems of probabilistic polynomial-time processes, with the sequential parts of each process written using the language described here. In this framework, the inherent complexity bounds allow us to quantify over all probabilistic polynomial-time adversaries by quantifying over processes expressible in the language. Related use of a language framework to quantify over adversaries has been developed in [1], but in a more abstract setting without complexity bounds.

2 Polynomial-time functionals

Nondeterministic and probabilistic Turing machines are usually defined as machines that may have more than one possible transition from a single configuration [19]. The difference between nondeterminism and randomness is not in the structure of the machine itself, but in the definition of acceptance: a nondeterministic machine accepts if there is *any* accepting computation, while probabilistic machines accept with probability determined by the number of coin flips along a computation path. It is easy to see that both forms of Turing machines are equivalent to deterministic Turing machines that use an oracle to decide which transition to take. Under the oracle-machine formulation, we would say that a "nondeterministic machine" accepts input x if there exists some oracle (representing all nondeterministic choices) that allows it to accept x. Similarly, we may regard a probabilistic machine as an oracle Turing machine that consults a randomly chosen oracle. Because of the corre-

spondence between branching computation and oracle computation, common complexity classes such as **NP**, **PP** and **BPP** are easily characterized using polynomial-time oracle computation. To be precise, we adopt the following definition:

Definition 1. A functional $f(\varphi, \vec{x})$, where φ may be any function from $\mathbb{N}$ to $\{0,1\}$, runs in *oracle polynomial rime* if there exists a polynomial p and an oracle Turing Machine M whose output with oracle φ and input $\vec{x}$ is $f(\varphi, \vec{x})$, and such that the running time of M on inputs $\vec{x}$ is bounded by $p(|\vec{x}|)$, where $|\vec{x}|$ is the vector $|x_1|, \ldots, |x_n|$ and $|x_i| = \lceil \log, x_i \rceil$.

It is important to notice that the running time of M must be bounded by a function of the length of the integer inputs, independent of the oracle. For this reason, the functions computable by oracle polynomial-time machines (as defined above) are different from the functions computable in polynomial time relative to any fixed oracle.

3 A language for oracle polynomial time

Our language OSLR is an extension of Hofmann's SLR with an oracle primitive. A central idea in SLR is that there are two types of natural number arguments to functions. Arguments of the first type, N, are bounded numeric values whose length (number of bits) can only be an additive constant above any of the input values. Since arguments of type N are bounded, it is **safe** to pass them to nested recursive functions. Arguments of the second type, ON, are *normal* natural number arguments that may be polynomially longer than input values of other functions. To avoid exponential-time computations, there are syntactic restrictions on primitive recursion that forbid use of normal arguments in the recursive position.

The types of SLR and OSLR are given by the grammar

τ	::=	**N**	(natural numbers)
		$\tau \to \tau$	(function from safe input)
		$\Box\tau \to \tau$	(functions from unsafe input)

It should be noted here that $\Box$ N is not actually an SLR type. There are two explanations for this situation, both equally valid. The simpler explanation is that Hofmann has modeled in a type system precisely what Bellantoni presented in a different framework. In particular, as explained in Section 5.1, Bellantoni makes use of input parameters of two sorts. There is only one sort of output in his framework, although there are syntactic restrictions on the places in which the output can be used. Hence, in SLR there is never a modality on the output type of a function. (The natural numbers are a special case; one may think of them as functions of no arguments.) The typing rules enforce the syntactic restrictions on composition.

Alternatively, we may employ an explanation derived from the theory of modal logics. In particular, the distinction between $\Box\tau$ and τ is related to modal operators. Originally inspired by type systems derived from linear logic [10], similar type distinctions have been used in program analysis and compilation to characterize the time at which a value becomes known [9, 20]. From the perspective of modal logic, there *do* exist modal output types, and consequently, there is a $\Box$ N type.

One contribution of [1 1] is the limited way that the modality $\Box$ may occur in types. This avoids the expression forms associated with ! in linear logic and, more generally associated with any modal type operator associated with any monad [IS]. A second innovation we adopt from [11] is a form of subtyping, with $A \to B <: \Box \quad A \to B$, further avoiding explicit conversions between types. Together, these innovations allow a useful form of type inference [l1]: there is a type-checking algorithm that can automatically determine the type of any expression, without requiring the distinction between N and $\Box$ lN to be written into expressions. (Since OSLR uses the same overall type system as SLR, this algorithm carries over to OSLR.) Thus, from the point of view of modal logic, the type Cl N exists, but the type-checking algorithm removes any need for its use.

The expressions of OSLR are given by the following grammar, where v may be any variable and τ any type:

e	::=	v	(variable)
		n	(numeral)
		$\mathbf{S_0}$ | $\mathbf{S_1}$	(doubling functions)
		$(e_1\ e_2)$	(application)
		fun$(v : \tau)\ e$	(abstraction)
		case, e_1 **zero** e_2 **even** e_3 **odd** e_4	(case distinction)
		saferec	(safe recursion))
		rand	(oracle bit)

Variables, lambda abstraction and application are standard from typed lambda calculus (see, e.g., [17]), with the modification that **fun**$(v : \tau)$ e may have

types $\tau \to \sigma$ or Or $\to \sigma$, according to the type inference algorithm [11]. In particular, a function gets the former type if and only if the argument of typτ is not passed to any function expecting a normal input. Functions So and $\mathbf{S_1}$ double a number or double and add 1, and **case,** has three branches, according to whether the first argument is zero, odd, or even. The restricted primitive recursion operator **saferec** is described below. The function **rand** returns the next bit from the oracle, with repeated calls potentially returning different bits. (There is nothing about this language that requires the oracle to be chosen randomly, but **we** use **rand** for oracle access since our primary interest is in probabilistic polynomial time.)

The type system is an extension of standard typed lambda calculus, with subtyping as described above and restrictions on computation achieved by careful distinction between N and ❑ lN in the typing of basic operations. The types of constants are as follows:

n	:	N, when n is an integer constant
$\mathbf{S_0}$	:	$\mathbf{N} \to \mathbf{N}$
$\mathbf{S_1}$	:	$\mathbf{N} \to \mathbf{N}$
case,	:	$\mathbf{N} \to \tau \to (\mathbf{N} \to \tau) \to (\mathbf{N} \to \tau) \to \tau$
saferec		❑
rand	:	**N**

Intuitively, we would expect n : ON for numeral n, since an explicit numeral has a fixed value, and therefore cannot implicitly define a fast-growing function of any input. However, ON itself is not a type. Instead, the typing rules of [11] are formulated so that it is possible to apply a function of type ON→ N to a numeral, since a numeral does not have any nonmodal free variables.

The intending meaning of the **case** construct is that

$$\textbf{case n zero } e_1 \textbf{ even } e_2 \textbf{ odd } e_3 = \begin{cases} e_1 & \text{if n = O} \\ e_2 n & \text{if n is even} \\ e_3 n & \text{if n is odd} \end{cases}$$

The intended meaning of **saferec** is that

$$\textbf{saferec } n\ a\ f = \begin{cases} a & \text{if n = 0} \\ f\ n\ (\textbf{saferec } \lfloor n/2 \rfloor \text{ a } f) & \text{otherwise} \end{cases}$$

The type of **saferec** captures the B^* requirements on predicative recursion described in Section 5.1. In particular, the output of a sub-recursion is presented to f in a safe position, i.e., as a N rather than a ON argument.

Note that **rand** has side-effects. (In particular, successive uses of **rand** return successive oracle bits, so **we** may view a use of **rand** as incrementing a counter that indicates which oracle bit should be used as the value of the next use of **rand.)** The presence of side-effects means that the order in which we perform reductions on terms can influence the value of those terms. For example, consider:

$$(\textbf{fun}(x : N)(\texttt{plus}\ x\ x))\ \textbf{rand}$$

where plus : ☐ N → ☐ N → N is the usual addition function. Under a "lazy" evaluation scheme, this term would evaluate to 1, if one of the next two oracle bits was 0 and the other was 1. We choose a LISP-like call-by-value evaluation scheme, feeling that it is the most "natural." Under our semantics, the term above can evaluate only to 0 or 2, but never to 1, because the evaluation of **rand** happens *before* the function application. A complete set-theoretic semantics for OSLR is given below.

It is worth mentioning one alternate language design that we considered. Instead of accessing an oracle bit-by-bit using **rand** : N, we could allow "random access" to the entire oracle by including a function **oracle** : N → N instead. At first glance, it might seem that the second is more general. However, it is easy to write a small loop that reads some polynomial number of oracle bits using **rand : N** and concatenates them into an integer value for later use. In contrast, we were not able to find any direct way translation in the opposite direction. Specifically, many randomized algorithms can be written fairly directly in OSLR using a "next random bit" primitive **rand** : N. When we attempted to find syntactic transformations that produced an equivalent algorithm using an oracle function **oracle** : N → N, we found that some artifacts **of** the type of **saferec** made it difficult to maintain a bit counter (indicating the next oracle bit to access) and pass this into and out of primitive recursive functions. We therefore decided to make a "next random bit" primitive **rand** : N a basic function of OSLR and prove that every function definable using **rand** is computable in polynomial time.

4 A Set-Theoretic Semantics for OSLR

We begin by defining a set-theoretic interpretation Set. $[\![\cdot]\!]$ on types. First, we define a mapping Set$[\![\cdot]\!]$ as

A probabilistic poly-time framework for protocol analysis

P. Lincoln*+
Computer Science Laboratory
SRI International

J. Mitchell*‡ M. Mitchell*§
Department of Computer Science
Stanford University

A. Scedrov* ¶
Department of Mathematics
University of Pennsylvania

Abstract

We develop a framework for analyzing security protocols in which protocol adversaries may be arbitrary probabilistic polynomial-time processes. In this framework, protocols are written in a form of process calculus where security may be expressed in terms of *observational* equivalence, a standard relation from programming language theory that involves quantifying over possible environments that might interact with the protocol. Using an asymptotic notion of probabilistic equivalence, we relate observational equivalence to polynomial-time statistical tests and discuss some example protocols to illustrate the potential of this approach.

1 Introduction

Protocols based on cryptographic primitives are commonly used to protect access to computer systems and to protect transactions over the internet. Two well-known examples are the Kerberos authentication scheme [15, 14], used to manage encrypted passwords, and the Secure Sockets Layer [12], used by internet browsers and servers to carry out secure internet transactions. Over the past decade or two, a variety of methods have been developed for analyzing and reasoning about such protocols. These approaches include specialized logics such as BAN logic [5], special-purpose tools designed for cryptographic protocol analysis [13], and theorem proving [26, 27] and model-checking methods using general purpose tools [16, 18, 23, 28, 29].

Although there are many differences among these approaches, most current approaches use the same basic model of adversary capabilities. This model, apparently derived from [10], treats cryptographic operations as "black-box" primitives. For example, encryption is generally considered a primitive operation, with plaintext and ciphertext treated as atomic data that cannot be decomposed into sequences of bits. In most uses of this model, as explained in [23, 26, 29], there are specific rules for how an adversary can learn new information. For example, if the decryption key is sent over the network "in the clear", it can be learned by the adversary. However, it is not possible for the adversary to learn the plaintext of an encrypted message unless the entire decryption key has already been learned. Generally, the adversary is treated as a nondeterministic process that may attempt any possible attack, and a protocol is considered secure if no possible interleaving of actions results in a security breach. The two basic assumptions of this model, perfect cryptography and nondeterministic adversary, provide an idealized setting in which protocol analysis becomes relatively tractable.

While there have been significant accomplishments using this model, the assumptions inherent in the standard model also make it possible to "verify" protocols that are in fact susceptible to attack. For example, the adversary is not allowed (by the model) to learn a decryption key by guessing it, since then some nondeterministic execution would allow a correct guess, and all protocols relying on encryption would be broken. However, in some real cases, adversaries can learn some bits of a key by statistical analysis, and can then exhaustively search the remaining (smaller) portion of the key space. Such an attack is simply not considered by the model described above, since it requires both knowledge of the particular encryption function involved and also the use of probabilistic methods.

Another way of understanding the limitations of common formal methods for protocol analysis is to consider the plight of someone implementing or installing a protocol. A protocol designer may design a protocol and prove that it is correct using the "black-box" cryptographic approach described above. However, an installed system must use a particular encryption function, or choice of encryption functions. Unfortunately, very few, if any, encryption functions satisfy all of the black-box assumptions. As a result, an implementation of a protocol may in fact be susceptible to attack, even though both the abstract protocol and the encryption function are individually correct.

Our goal is to establish an analysis framework that can be used to explore interactions between protocols and cryptographic primitives. In this paper, we set the stage for a form of protocol analysis that allows the analysis of these interactions as well as many other attacks not permitted in the standard model. Our framework uses a language for defining communicating probabilistic polynomial-time processes [22]. We restrict processes to probabilistic polynomial time so that we can say that a protocol is secure if there is

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505.
+Additional support from NSF CCR-9509931.
*Additional support from NSF CCR-9629754.
§Additional support from Stanford University Fellowship.
¶Additional support from NSF Grant CCR-9800785.

no definable program which, when run in parallel with the protocol, causes a security breach. Establishing a bound on the running time of an adversary allows us to lift other restrictions on the behavior of an adversary. Specifically, an adversary may send randomly chosen messages, or perform sophisticated (yet probabilistic polynomial-time) computation to derive an attack from statistical analysis of messages overheard on the network. In addition, we treat messages as sequences of bits and allow specific encryption functions such as RSA or DES to be written in full as part of a protocol. An important feature of our framework is that we can analyze probabilistic as well as deterministic encryption functions and protocols. Without a probabilistic framework, it would not be possible to analyze an encryption function such as ElGamal [11], for example, for which a single plaintext may have more than one ciphertext.

In our framework, following the work of Abadi and Gordon [1], security properties of a protocol P may be formulated by writing an idealized protocol Q so that, intuitively, for any adversary M, the interactions between M and P have the same observable behavior as the interactions between M and Q. Following [1], this intuitive description may be formalized by using observational equivalence (also called observational congruence), a standard notion from the study of programming languages. Namely, two processes (such as two protocols) P and Q are observationally equivalent, written $P \simeq$ Q, if any program $\mathcal{C}[P]$ containing P has the same observable behavior as the program C[Q] with Q replacing P. The reason observational equivalence is applicable to security analysis is that it involves quantifying over all possible adversaries, represented by the environments, that might interact with the protocol participants. Our framework is a refinement of this approach in that in our asymptotic formulation, observational equivalence between probabilistic polynomial-time processes coincides with the traditional notion of indistinguishability by polynomial-time statistical tests [17, 30], a standard way of characterizing cryptographically strong pseudo-random number generators.

2 A language for protocols and intruders

2.1 Protocol description

A protocol consists of a set of programs that communicate over some medium in order to achieve a certain task. In this paper, we are concerned with the security of cryptographic protocols, which are protocols that use some set of cryptographic operations. For simplicity, we will only consider protocols that require some fixed number of communications per instance of the protocol. For example, for each client-server session, we assume that there is some fixed number of client-server messages needed to execute the protocol. This is the case for most handshake protocols, key-exchange protocols and authentication protocols, such as Kerberos, the Secure Sockets Layer handshake protocol, and so on. While we do not foresee any fundamental difficulty in extending our basic methods to more general protocols that do not have a fixed bound set in advance, there are some technical complications that we avoid by making this simplifying assumption.

We will use a form of π-calculus (a general process calculus) [21] for defining protocols. One reason for using a precise language is to make it possible to define protocols exactly. As will be illustrated by example, many protocols have been described using an imprecise notation that describes possible traces of the protocol, but does not define the way that protocol participants may respond to incorrect messages or other communication that may arise from the intervention of a malicious intruder. In contrast, process calculus descriptions specify the response to adversary actions precisely.

The second reason for defining a precise process computation and communication language is to characterize the possible behavior of a malicious intruder. Specifically, we assume that the protocol adversary may be any process or set of processes that are definable in the language. In the future, we hope to follow the direction established by the spi-calculus [1] and use proof methods for forms of observational congruence. However, in order to proceed in this direction, we need further understanding of probabilistic observational congruence and approximations such as probabilistic bisimulation. Since there has been little prior work on probabilistic process formalisms, one of our near-term goals is to better understand the forms of probabilistic reasoning that would be needed to carry out more accurate protocol analysis.

2.2 Protocol language

The protocol language consists of a set of *terms*, or sequential expressions that do not perform any communication, and processes, which can communicate with one another. The process portion of the language is a restriction of standard r-calculus. All computation done by a process is expressed using terms. Since our goal is to model probabilistic polynomial-time adversaries by quantifying over processes definable in our language, it is essential that all functions definable by terms lie in probabilistic polynomial time.

Although we use pseudo-code to write terms in this paper, we have developed an applied, simply-typed lambda calculus which exactly captures the probabilistic polynomial-time terms. Our language is described in [22].

2.3 Processes

For any set of terms, we can define a set of processes. Since we are interested in protocols with a fixed number of steps, we do not need arbitrary looping. We therefore use a bounded subset of asynchronous π-calculus, given by the following grammar:

$P ::=$	
$\mathcal{O}$	empty process (does nothing)
$\overline{n}\langle M\rangle$	transmit value of M on port n
$n(x).\ P$	read value for x on port n and do P
$P \mid Q$	do P in parallel with Q
$\nu n.P$	do P with port n considered private
$!_k P$	execute up to k copies of process P
$[M = N]P$	if M = N then do P (guarded command)
let x = M in P	bind variable x to M and do P

2.4 Communication

Intuitively, the communication medium for this language is a buffered network that allows messages sent by any process to be received by any other process, in any order. Messages are essentially pairs consisting of a "port name" and a data value. The expression $\overline{n}\langle M\rangle$ sends a message M on the port n. In other words, it places a pair $\langle n, M\rangle$ onto the

network. The expression n(z). P matches any pair (n, m) and continues process P with x bound to value m. When n(x).P matches a pair (n, ***M)*** , the pair (n, ***M)*** is removed from the network and is no longer available to be read by another process. Evaluation of n(z). P does not proceed unless or until a pair $\langle n, \mathrm{m}\rangle$ is available.

Although we use port names to indicate the intended source and destination of a communication, there are no delivery guarantees in this model. Any process containing a read expression for a given port can read any message sent by any other process on that port. In particular, an adversary can read any public network message sent by any protocol participant.

Some readers may wonder why reading a message has the side-effect of removing it from the network. One reason is that we wish to allow an attacker to intercept messages without forwarding them to other parties. This may occur in practice when an attacker floods the subnet of a receiver. In addition, we may express passive reads, which do not remove messages from the network, as a combination of destructive read and resend. To make this precise, let us write $n_{pasv}(x).\ P$ as an abbreviation for n(x). $(\overline{n}\langle x\rangle \mid P)$. It is not hard to see that this definable combination of actions is equivalent to the intuitive notion of a passive read. For example, consider the process $\overline{x}\langle a\rangle \mid n_{pasv}(x).\ P \mid Q$ containing an output and a passive read. If the passive read is scheduled first, one computation step of this process leads to $\overline{x}\langle a\rangle \mid P[a/x] \mid$ Q which is what one would expect from a passive read primitive. Further details on the operational semantics of the process language appear in Appendix A.

2.5 Example using symbolic cryptosystem

For readers not familiar with π-calculus, we give a brief example using a simple set of terms with "black-box" cryptography. Specifically, for this section only, let us use algebraic expressions over sorts plain, cipher and key, representing plaintext, ciphertext and keys, and function symbols

$$\text{encrypt: plain x } key \rightarrow cipher$$
$$decrypt\text{: } cipher \text{ x } key \rightarrow \text{plain}$$

We illustrate the calculus by restating a simple protocol written in "the notation commonly found in the literature" where $A \rightarrow B$ indicates a message from ***A*** to B.

In the following protocol, A sends an encrypted message to B. After receiving a message back that contains the original plaintext, A sends another message to ***B.***

$$\begin{array}{lll} A \rightarrow \boldsymbol{B}\text{:} & \text{encrypt}\,(p_1, k_B) & (1)\\ B \rightarrow A\text{:} & encrypt(conc(p_1, p_2),\ k_A) & (2)\\ A \rightarrow \boldsymbol{B}\text{:} & encrypt(p_3, k_B) & (3) \end{array}$$

We can imagine that p_1 is a simple message like "hello" and p_3 is something more critical, like a credit card number. Intuitively, after ***A*** receives a message back containing p_1, A may believe that it is communicating with ***B*** because only ***B*** can decrypt a message encoded with ***B's*** key k_B

This protocol can be written in π-calculus using the same cryptographic primitives. However, certain decisions must be made in the translation. Specifically, the notation above says what communication will occur when everything goes right, but does not say how the messages depend on each other or what might happen if other messages are received. Here is one interpretation of the protocol above. In this interpretation, ***B*** responds to A without examining the contents of the message from ***A*** to ***B.*** However, in step 3, ***A*** only responds to ***B*** if the message it receives is exactly the encryption of the concatenation of p_1 and p_2

$$\begin{array}{ll} \overline{AB}\langle encrypt(p_1, k_B)\rangle & (1)\\ \mid\ AB(x).\ \overline{BA}\langle encrypt(conc(decrypt(x, K_B), p_2), k_A)\rangle & (2)\\ \mid\ BA(y).\ [decrypt(y, K_A) = conc(p_1, p_2)] & \\ \quad \overline{AB}\langle encrypt(p_3, k_A)\rangle & (3) \end{array}$$

In words, the protocol is expressed as the parallel composition of three processes. Port AB is used for messages from ***A*** to ***B*** while port BA for messages from ***B*** to ***A.***

A fundamental idea that we have adopted from spi-calculus [1] is that an intruder may be modeled by a process context, which is a process expression containing a hole indicating a place that may be filled by another process. Intuitively, we think of the context as the environment in which the process in the hole is executed. To give a specific example, consider the context

$$\mathcal{C}[\ ] = [\ ] \mid AB(x).\ \overline{AB}\langle encrypt(p_1, k_C)\rangle$$

where the empty square brackets [ ] indicate the hole for an additional process. If we insert a process P in this context, the resulting process $\mathcal{C}[P]$ will run $AB(x).\ \overline{AB}\langle encrypt(p_1,\ k_C)\rangle$ in parallel with P. It is easy to see that if we insert the protocol above in this context, then the context could intercept the first message from ***A*** to ***B*** and replace it by another one using a different key.

2.6 Example

Our first example (continued in Section 4.1) is a simple protocol based on ElGamal Encryption [11] and Diffie-Hellman Key Exchange [8], formulated in a way that gives us a series of steps to look at. The protocol assumes that a prime p and generator g of $\mathcal{Z}_p^*$ are given and publicly available. Using the notation commonly found in the security literature, this protocol may be written

$$\begin{array}{lllll} \boldsymbol{A} & \rightarrow & \boldsymbol{B} & : & g^a \mod p\\ \boldsymbol{B} & \rightarrow & \boldsymbol{A} & : & g^b \mod p\\ \boldsymbol{A} & \rightarrow & \boldsymbol{B} & : & \mathsf{msg} * g^{ab} \mod p \end{array}$$

The main idea here is that by choosing a and receiving $g^b \mod p$, Alice can compute $g^{ab} \mod p$. Bob can similarly compute $g^{ab} \mod p$, allowing Alice and Bob to encrypt by multiplying by g^{ab} and decrypt by dividing by g^{ab}. It is generally believed that no eavesdropper can compute $g^{ab} \mod p$ by overhearing g^a and g^b Since this protocol is susceptible to attack by an adversary who intercepts a message and replaces it, we will only consider adversaries who listen passively and try to determine if the message msg has been sent.

In π-calculus notation, the protocol may be written as follows. We use the convention that port $\overline{AB}_i$ is used for the ith message from ***A*** to ***B,*** and meta-notation for terms that could be written out in detail in our probabilistic polynomial-time language. To make explicit the assumption that p and g are public, the protocol transmits them on a public port.

let p be a random n-bit prime and
  g a generator of $\mathcal{Z}_p^*$
in $\overline{PUBLIC}\langle p\rangle \mid \overline{PUBLIC}\langle g\rangle$
| let a be a random number in $[1, p-1]$

$$\begin{array}{l} \quad \text{in } \overline{AB_1}\langle g^a \bmod p\rangle \\ \quad\quad | \; B_A(x).\, \overline{AB_2}\langle \mathsf{msg} * x^a \bmod p\rangle \\ | \quad \text{let b be a random number in } [1, p-1] \\ \quad \text{in } A_{B_1}(y).\, \overline{BA}\langle g^b \bmod p\rangle \end{array}$$

An analysis appears in Section 4.1.

2.7 Parallelism, Nondeterminism and Complexity

For complexity reasons, we must give a nonstandard probabilistic semantics for to parallel composition. Specifically, our intention is to design a language of communicating processes so that an adversary expressed by a set of processes is restricted to probabilistic polynomial time. However, if we interpret parallel composition in the standard nondeterministic fashion, then a pair of processes may nondeterministically "guess" any secret information.

This issue may be illustrated by example. Let us assume that B has a private key K_b that is k bits long and consider the one-step protocol where A encrypts a message using this key and sends it to B.

$$A \;\rightarrow\; B \;:\; \{\mathsf{msg}\}_{K_b}$$

We assume that an evil adversary wishes to discover the message **msg.** If we allow the adversary to consist of 3 processes E_0, E_1 and E, scheduled nondeterministically, then this can be accomplished. Specifically, we let

$$\begin{array}{lcl} A & = & \overline{AB}\langle encrypt(K_b,\ \mathsf{msg})\rangle \\ E_0 & = & !_k\, \overline{E}\langle 0\rangle \\ E_1 & = & !_k\, \overline{E}\langle 1\rangle \\ E & = & {}_E(b_0).\ \ {}_E(b_{k-1}).\ \ {}_{AB}(x). \\ & & \overline{Public}\langle\ decrypt(\ conc(b_0, . , b_{k-1}),\ \mathbf{msg})\mathbf{)} \end{array}$$

Adversary processes E_0 and E_1 each send k bits to E, all on the same port. Process E reads the message from A to B, concatenates the bits that arrive nondeterministically in some order, and decrypts the message. One possible execution of this set of processes allows the eavesdropper to correctly decrypt the message. Under traditional nondeterministic semantics of parallel composition, this means that such an eavesdropper can break any encryption mechanism.

Intuitively, the attack described above should not succeed with much more than probability $1/2^k$, the probability of guessing key K_b using random coins. Specifically, suppose that the key K_b is chosen at random from a space of order 2" keys. If we run processes E_0, E_1, E on physical computers communicating over an ethernet, for example, then the probability that communication from E_0 and E_1 will accidentally arrive at E in an order producing exactly K_b cannot be any higher than the probability of randomly guessing K_b. Therefore, although nondeterminism is a useful modeling assumption in studying correctness of concurrent programs, it does not seem helpful for analyzing cryptographic protocols.

Since nondeterminism does not realistically model the probability of attack, we use a probabilistic form of parallel composition. This is described in more detail in Appendix A, which contains a full operational semantics.

3 Process Equivalence

Observational equivalence, also called observational congruence, is a standard notion in the study of programming languages. We explain the general concept briefly, as it arises in a variety of programming languages.

The main idea is that the important features of a part of a program, such as a function declaration, processes or abstract data type, are exactly those properties that can be observed by embedding them in full programs that may produce observable output. To formalize this in a specific programming language $\mathcal{L}$, we assume the language definitions gives rise to some set of program contexts, each context $\mathcal{C}[\,]$ consisting of a program with a "hole" (indicated by empty square brackets [ ]) to insert a phrase of the language, and some set *Obs* of concrete observable actions, such as integer or string outputs. We also assume that there is some semantic evaluation relation $\overset{eval}{\rightsquigarrow}$, with $M \overset{eval}{\rightsquigarrow} v$ meaning that evaluation or execution of the program M produces the observable action v. In a functional language, this would mean that v is a possible value of M, while in a concurrent setting this might mean that v is a possible output action. Under these assumptions, we may associate an *experiment* on program phrase with each context C[ ] and observable v: given phrase P, run the program $\mathcal{C}[P]$ obtained by placing P in the given context and see whether observable action v occurs. The main idea underlying the concept of observational equivalence is that the properties of a program phrase that matter in program construction are precisely the properties that can be observed by experiment. Phrases that give the same experimental results can be considered equivalent.

Formally, we say program phrases P and Q are *observationally* equivalent, written $P \simeq Q$, if, for all program contexts C[ ] and observables $v \in \mathcal{O}$, we have

$$\mathcal{C}[P] \overset{eval}{\rightsquigarrow} v \text{ iff } \mathcal{C}[Q] \overset{eval}{\rightsquigarrow} \text{v}$$

In other words, $P \simeq$ Q if, for any program $\mathcal{C}[P]$ containing P, we can make exactly the same concrete observations about the behavior of $\mathcal{C}[P]$ as we can about the behavior of the program C[Q] obtained by replacing some number of occurrences of P by Q.

For the process language considered in this paper, we are interested in contexts that distinguish between processes. (We will not need to consider observational equivalence of terms.) Therefore, the contexts of interest are process expressions with a "hole", given by the following grammar

$$\begin{array}{lcl} \mathcal{C}[\,] & ::= & [\,] \;|\; n(x).\mathcal{C}[\,] \;|\; P|\mathcal{C}[\,] \;|\; \mathcal{C}[\,]|Q \;| \\ & & \nu n.\mathcal{C}[\,] \;|\; [M=N]\mathcal{C}[\,] \;|\; \mathsf{let}\ x = M\ \mathsf{in}\ \mathcal{C}[\,] \end{array}$$

A process observation will be a communication event on a port whose name is not bound by ν. More specifically, we let *Obs* be the set of pairs $\langle n, m\rangle$, where n is a port name and **m** is an integer, and write $P \overset{eval}{\rightsquigarrow} \langle n, m\rangle$ if evaluation of process expression P leads to a state (represented by a process expression) of the form $|\overline{n}\langle m\rangle$ in which the process is prepared to communicate integer m on port **n** and n is not within the scope of a binding νn. (This can be made more precise using the structural equivalence relation in the Appendix.) In more general terms, $P \overset{eval}{\rightsquigarrow} v$ in our language if process P publicly outputs v.

The general definition of $\simeq$ above is essentially standard for deterministic or nondeterministic functional, imperative or concurrent languages. Some additional considerations enter when we consider probabilistic languages. Drawing from standard notions in cryptography, we propose the following adaptation of observational equivalence to the probabilistic polynomial-time process language at hand.

Intuitively, given program phrases P and Q, context C[ ] and observable action v, it seems reasonable to compare the

Linköping Electronic Articles in
Computer and Information Science
Vol. 2(1998): nr 1

The Complexity of Model Checking in Modal Event Calculi with Quantifiers

Iliano Cervesato
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
iliano@cs.stanford.edu

Massimo Franceschet **Angelo Montanari**
Dipartimento di Matematica e Informatica
Università di Udine
Via delle Scienze, 206 – 33100 Udine, Italy
{*francesc*|*montana*}*@dimi.uniud.it*

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1998/001/

Abstract

Kowalski and Sergot's Event Calculus (EC) is a simple temporal formalism that, given a set of event occurrences, derives the maximal validity intervals (MVIs) over which properties initiated or terminated by these events hold. It does so in polynomial time with respect to the number of events. Extensions of its query language with Boolean connectives and operators from modal logic have been shown to improve substantially its scarce expressiveness, although at the cost of an increase in computational complexity. However, significant sublanguages are still tractable. In this paper, we further extend *EC* queries by admitting arbitrary event quantification. We demonstrate the added expressive power by encoding a hardware diagnosis problem in the resulting calculus. We conduct a detailed complexity analysis of this formalism and several sublanguages that restrict the way modalities, connectives, and quantifiers can be interleaved. We also describe an implementation in the higher-order logic programming language $\lambda Prolog$.

1 Introduction

The *Event* Calculus, abbreviated *EC* [9], is a simple temporal formalism designed to model and reason about scenarios characterized by a set of *events,* whose occurrences have the effect of starting or terminating the validity of determined properties. Given a *possibly incomplete* description of when these events take place and of the properties they affect, *EC* is able to determine the maximal *validity intervals,* or *MVIs*, over which a property holds uninterruptedly. In practice, since this formalism is usually implemented as a logic program, *EC* can also be used to check the truth of *MVIs* and process boolean combinations of *MVI* verification or computation requests. The range of queries that can be expressed in this way is however too limited for modeling realistic situations.

A systematic analysis of *EC* has recently been undertaken in order to gain a better understanding of this calculus and determine ways of augmenting its expressive power. The keystone of this endeavor has been the definition of an extendible formal specification of the functionalities of this formalism [3]. This has had the effects of establishing a semantic reference against which to verify the correctness of implementations [4], of casting *EC* as a model checking problem [5], and of setting the ground for studying the complexity of this problem, which was proved polynomial [2]. Extensions of this model have been designed to accommodate constructs intended to enhance the expressiveness of *EC.* In particular, modal versions of *EC* [1], the interaction between modalities and connectives [5], and preconditions [6] have all been investigated in this context.

In this paper, we continue this endeavor to enhance the expressive power of *EC* by considering the possibility of quantifying over events in queries, in conjunction with boolean connectives and modal operators. We also admit requests to check the relative order of two events. We thoroughly analyze the representational and computational features of the resulting formalism, that we call *QCMEC.* We also consider two proper sublanguages of it, *EQCMEC,* in which modalities are applied to atomic formulas only, and *CMEC,* which is quantifier-free. We show that *QCMEC* and its restrictions can effectively be used to encode diagnosis problems. Moreover, we provide an elegant implementation in the higher-order logic programming language $\lambda Prolog$ [10] and prove its soundness and completeness. As far as computational complexity is concerned, we prove that model checking in *CMEC, EQCMEC,* and *QCMEC* is PSPACE-complete. However, while solving an *EQCMEC* problem is exponential in the size of the query, it has only polynomial cost in the number n of events, thus making *EQCMEC* a viable formalism for *MVI* verification or computation. Since in most realistic applications the size of databases (n) dominates by several orders of magnitude the size of the query, n is asymptotically the parameter of interest.

The main contributions of this work are: (1) the extension of a family of modal event calculi with quantifiers; (2) permitting queries to mention ordering information; (3) the use of the higher-order features of modern logic programming languages in temporal reasoning; and (4) analyzing the complexity of model checking in these extensions of *EC.*

This paper is organized as follows. In Section 2, we formalize *QCMEC* and significant subcalculi. Section 3 exemplifies how this calculus can adequately model certain hardware diagnosis problems. In Section 4, we briefly introduce the logic programming language $\lambda Prolog$, give an implementation of *QCMEC* in it and prove the soundness and completeness of the resulting

program. We study the complexity of *QCMEC* and its sublanguages in Section 5. We outline directions of future work in Section 6.

2 Modal Event Calculi with Quantifiers

In this section, we first briefly recall the syntax and semantics of a number of modal event calculi. We invite the interested reader to consult [1, 3, 5, 8, 9] for motivations, examples, properties, and technical details. We then extend these basic definitions to give a semantic foundation to refinements of these calculi with quantifiers.

2.1 Event Calculus

The Event Calculus *(EC)* [9] and the extensions we propose aim at modeling scenarios that consist of a set of events, whose occurrences over time have the effect of initiating or terminating the validity of properties, some of which may be mutually exclusive. We formalize the time-independent aspects of a situation by means of an *EC-structure* [1], defined as follows:

Definition 2.1 *(EC-structure)*

A structure for *the* Event Calculus (or EC-structure) *is a quintuple* $\mathcal{H} = (E, P, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ *such that:*

- $E = \{e_1, \ldots, e_n\}$ *and* $P = \{p_1, \ldots, p_m\}$ *are finite sets of* events and properties, *respectively.*
- $[\cdot\rangle : P \to \mathbf{2}^E$ *and* $\langle\cdot] : P \to \mathbf{2}^E$ *are respectively the* initiating *and* terminating map *of* $\mathcal{H}$. *For every property* $p \in P$, $[p\rangle$ *and* $(p]$ *represent the set of events that initiate and terminate p, respectively.*
- $]\cdot,\cdot[\subseteq P \times P$ *is an irreflexive and symmetric relation, called the* exclusivity relation, *that models exclusivity among properties.* □

As in the original EC paper [9], we define the initiating and terminating maps in terms of event occurrences rather than event types. The latter approach can however easily be accomodate in our setting.

The temporal aspect of *EC* is given by the order in which events happen. Unlike the original presentation [9], we focus our attention on situations where the occurrence time of events is unknown and only assume the availability of incomplete information about the relative order in which they have happened. We however require the temporal data to be consistent so that an event cannot both precede and follow some other event. Therefore, we formalize the time-dependent aspect of a scenario modeled by *EC* by means of a (strict) *partial order,* i.e. an irreflexive and transitive relation, over the involved set of event occurrences. We write $W_{\mathcal{H}}$ for the set of all partial orders over the set of events E in an EC-structure $\mathcal{H}$, use the letter w to denote individual orderings, or *knowledge states,* and write $e_1 <_w e_2$ to indicate that e_1 precedes e_2 in w. The set $W_{\mathcal{H}}$ of all knowledge states naturally becomes a reflexive ordered set when considered together with the usual subset relation $\subseteq$, which is indeed reflexive, transitive and antisymmetric. An *extension* of a knowledge state w is any element of $W_{\mathcal{H}}$ that contains w as a subset. We write $\mathrm{Ext}_{\mathcal{H}}(w)$ for the set of all extensions of the ordering w in $W_{\mathcal{H}}$.

Given a structure $\mathcal{H} = (E, P, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ and a knowledge state w, *EC* permits inferring the maximal *validity intervals,* or *MVIs*, over which a property p holds uninterruptedly. We represent an *MVI* for p as $p(e_i, e_t)$,

Complexity of Problems on Graphs Represented as OBDDs*

J. Feigenbaum,[1] S. Kannan,[2] ** M. Y. Vardi,[3] *** M. Viswanathan[2] †

[1] AT&T Labs – Research
Room C203, 180 Park Avenue
Florham Park, NJ 07932 USA
jf@research.att.com
[2] Computer and Information Sciences
University of Pennsylvania
Philadelphia, PA 19104 USA
kannan@central.cis.upenn.edu
maheshv@gradient.cis.upenn.edu
[3] Computer Science
Rice University
Houston, TX 77251 USA
vardi@cs.rice.edu

Abstract. To analyze the complexity of decision problems on graphs, one normally assumes that the input size is polynomial in the number of vertices. Galperin and Wigderson [GW83] and, later, Papadimitriou and Yannakakis [PY86] investigated the complexity of these problems when the input graph is represented by a polylogarithmically succinct circuit. They showed that, under such a representation, certain trivial problems become intractable and that, in general, there is an exponential blow up in problem complexity. Later, Balcázar, Lozan, and Torán [Bal96,BL89,BLT92,Tor88] extended these results to problems whose inputs were structures other than graphs.

In this paper, we show that, when the input graph is represented by a ordered binary decision diagram (OBDD), there is an exponential blow up in the complexity of most graph problems. In particular, we show that the GAP and AGAP problems become complete for PSPACE and EXP, respectively, when the graphs are succinctly represented by OBDDs.

---

* An extended abstract of this paper appears in the Proceedings of the 1998 Symposium on Theoretical Aspects of Computer Science.

** Work done in part as a consultant to AT&T and supported in part by NSF grant CCR96-19910 and ONR Grant N00014-97-1-0505.

*** Work done as a visitor to DIMACS and Bell Laboratories as part of the DIMACS Special Year on Logic and Algorithms and supported in part by NSF grants CCR-9628400 and CCR-9700061 and by a grant from the Intel Corporation.

† Supported by grants NSF CCR-9415346, NSF CCR-9619910, AFOSR F49620-95-1-0508, ARO DAAH04-95-1-0092, and ONR Grant N00014-97-1-0505.

1 Introduction

The efficiency of algorithms is generally measured as a function of input size [CLR89]. In analyses of graph-theoretic algorithms, graphs are usually assumed to be represented either by adjacency matrices or by adjacency lists. However, many problem domains, most notably computer-aided verification [Bry86,BCM+92,Kur94a], involve extremely large graphs that have regular, repetitive structure. This regularity can yield very succinct encodings of the input graphs, and hence one expects a change in the time- or space-complexity of the graph problems.

The effect of succinct input representations on the complexity of graph problems was first formalized and studied by Galperin and Wigderson [GW83]. They discovered that, when adjacency matrices are represented by polylogarithmically-sized circuits, many computationally tractable problems become intractable. Papadimitriou and Yannakakis [PY86] later showed that such representations generally have the effect of exponentiating the complexity (time or space) of graph problems. Following this line of research, Balcázar, Lozano, and Torán [Bal96,BL89,BLT92,Tor88] extended these results to problems whose inputs were structures other than graphs and provided a general technique to compute the complexity of problems with inputs represented by succinct circuits [BLT92]. They also provide sufficiency conditions for problems that become intractable when inputs are represented in this way. Veith [Vei95,Vei96] showed that, even when inputs are represented using Boolean formulae (instead of circuits), a problem's computational complexity can experience an exponential blow-up. He also provides sufficient conditions for when the problems become hard.

The possibility of representing extremely large graphs succinctly has attracted a lot of attention in the area of computer-aided verification [Bry86,BCM+92,Kur94a]. In this domain, graphs are represented by *ordered binary decision diagrams* (OBDDs). OBDDs are special kinds of rooted, directed acyclic graphs that are used to represent Boolean formulae. Because of their favorable algorithmic properties, they are widely used in the areas of digital design, verification, and testing [Bry92,BCM+92,McM93]. Experience has shown that OBDD-based algorithmic techniques scale up to industrial-sized designs [CGH+95], and tools based on such techniques are gaining acceptance in industry [BBDG+94]. Although OBDDs provide canonical succinct representations in many practical situations, they are exponentially less powerful than Boolean circuits in the formal sense that there are Boolean functions that have polynomial-sized circuit representations but do not have subexponential-sized OBDD representations [Pon95a,Pon95b]. (On the other hand, the translation from OBDDs to Boolean circuits is linear [Bry86].) Thus, the results of [BL89,BLT92,GW83,PY86,Tor88,Vei95,Vei96] do not apply to OBDD-represented graphs. Furthermore, even though Boolean formulae are, in terms of representation size, less powerful than circuits, they are still more succinct than OBDDs. Translation from OBDDs to formulae leads to at most a quasi-polynomial $(n^{\log n})$ blow-up, whereas there are functions (e.g., multiplication of binary integers) that have polynomial-sized formulae but require exponential-sized OBDDs. Indeed, while the satisfiability problem is NP-

complete for Boolean formulae, it is in nondeterministiclogspace for OBDDs [Bry86]. Therefore, the results in [Vei95,Vei96] do not apply to our case.

In this paper, we show that, despite these theoretical limitations on the power of OBDDs to encode inputs succinctly, using them to represent graphs nonetheless causes an exponential blow-up in problem complexity. That is, the well-studied phenomenon of exponential increase in computational complexity for graph problems with inputs represented by Boolean circuits or formulae [BL89,BLT92,GW83,PY86,Tor88,Vei95,Vei96] also occurs when the graphs are represented by OBDDs. Graph properties that are ordinarily NP-complete become NEXP-complete. The Graph Accessibility Problem (GAP) and the Alternating Graph Accessibility Problem (AGAP) for OBDD-encoded graphs are PSPACE-complete and EXP-complete, respectively. Both GAP and AGAP are important problems in *model checking*, a domain in which OBDDs are widely used [BCM+92,EL86,KV96,Kur94b].

In section 2, we formally define OBDDs and present some known results about them. In section 3, we discuss the problem in greater detail and compare Papadimitriou and Yannakakis's result to ours. Finally, in sections 5-7, we give our technical results.

2 Preliminaries

Definition 1. A Binary *Decision* Diagram (BDD) is a single-rooted, directed acyclic graph in which

- Each internal node (i.e., a node with nonzero outdegree) is labeled by a Boolean variable.
- Each internal node has outdegree 2. One of the outgoing edges is labeled 1 (the "then-edge") and the other is labeled 0 (the "else-edge").
- Each external node (i.e., a node with zero outdegree) is labeled 0 or 1.

Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of Boolean variables that occur as labels of nodes in a given BDD *B*. Each assignment $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ of Boolean values to these variables naturally defines a computational path– the one that leads from the root to an external node and has the property that, when it reaches a node labeled x_i, it follows the edge labeled α_i, for any i.

Definition 2. A BDD *B represents the* Boolean *function* $f(x_1, x_2, \ldots, x_n)$ if, for each assignment $\alpha = (\alpha_1, \alpha_2, \ldots, a,)$ to the variables of f, the computation path defined by α terminates in an external node that is labeled by the value $f(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

Definition 3. Two nodes u and v of a BDD are *equivalent* if the BDD rooted at u and v represent the same boolean function. A BDD in which no two different nodes are equivalent is called *reduced*.

Definition 4. Let < be a total ordering on a set X. An *Ordered Binary Decision Diagram* (OBDD) over (X, <) is a reduced BDD with node-label set X such that, along any path from the root to an external node, there is at most one occurrence of each variable, and the order in which the variables occur along the path is consistent with the order (X, <). The size of an OBDD is the number of internal nodes in it.

Definition 5. An OBDD 0 *represents* the graph G = *(V,* E) if 0 represents the Boolean function adj, where

$$adj(v_1, v_2) = \begin{cases} 1 \text{ if and only if } < v_1, v_2 > \in E \\ 0 \text{ otherwise} \end{cases}$$

Theorem 6 (Bryant [Bry86]). *For each Boolean function f and ordering (X, <) of the set of variables X, there is a unique (up to isomorphism) OBDD over (X, <) that represents f.*

Theorem 7 (Bryant [Bry86]). *Let F and G be OBDDs over (X, <) representing functions f and g, respectively. Let the size of F be m, the size of G be n,* and $< op >$ *be any Boolean operation. Then there is an OBDD* over $(X, <)$ *of size at most mn and constructable in time polynomial* in *m and n that represents* $f < op > g$.

Definition 8. Let L = (G, <) be a linear order on the gates of a circuit, where the inputs and outputs are classified as special instances of gates. We say that the *forward* cross *section of the circuit at gate* g is the set of wires connected to the output of some gate g_1 and an input of some gate g_2 such that $g_1 \leq$ g and $g < g_2$. The *reverse cross section of the circuit at gate g* is the set of wires connected to an output of some gate g_1 and an input of some gate g_2 such that $g_2 \leq g$ and $g < g_1$.

Definition 9. The *forward width of a circuit under order L,* denoted w_f, is the maximum, over all gates g, of the forward cross section at g. Similarly, *the reverse width of the circuit under order L,* denoted by w_r, is the maximum, over all gates g, of the reverse cross section at g.

Theorem 10 (Berman [Ber89]). *For a circuit* and *gate-ordering with* $w_r = 0$, *there exists a variable* ordering *such that the OBDD size is bounded by* $n2^{w_f}$, *where n is the number of inputs to the circuit.*

Notation: We will be interested in complexity classes C that have universal Turing machines and complete problems. Let $U_{\mathbf{C}}$ denote the Universal Turing machine for the complexity class C. Let $\mathcal{L}(U_{\mathbf{C}})$ be the language accepted by the machine $U_{\mathbf{C}}$ i.e., $\mathcal{L}(U_{\mathbf{C}})$ = { x encodes a C-bounded Turing machine M and an input y such that M accepts y}.

For an n-bit number x, we will refer to the i^{th} bit by $x^{(i)}$, where $x^{(n)}$ is the most significant bit.

Optimization Complexity of Linear Logic Proof Games

Patrick D. Lincoln* John C. Mitchell+ Andre Scedrov‡

Abstract

A class of linear logic proof games is developed, each with a numeric score that depends on the number of preferred axioms used in a complete or partial proof tree. The complexity of these games is analyzed for the NP-complete multiplicative fragment (MLL) extended with additive constants and the PSPACE-complete multiplicative, additive fragment (MALL) of propositional linear logic. In each case, it is shown that it is as hard to compute an approximation of the best possible score as it is to determine the optimal strategy. Furthermore, it is shown that no efficient heuristics exist unless there is an unexpected collapse in the complexity hierarchy.

*lincoln@csl.sri.com SRI International Computer Science Laboratory, Menlo Park CA 94025 USA. Work supported under NSF Grant CCR-9224858 and ONR Grant N00014-95-C-0168.

†mitchell@cs.stanford.edu WWW: http://theory.stanford.edu/people/jcm/home.html Department of Computer Science, Stanford University, Stanford, CA 94305-9045 USA. Partially supported by NSF Grants CCR-9303099 and CCR-9629754.

‡scedrov@cis.upenn.edu WWW: http://www.cis.upenn.edu/~scedrov Department of Mathematics, University of Pennsylvania, Philadelphia, PA 19104-6395 USA. Partially supported by NSF Grant CCR-94-00907, by ONR Grant N00014-92-J-1916, and by a Centennial Research Fellowship from the American Mathematical Society.

1 Introduction

Linear logic, introduced in [12], is a refinement of classical logic often described as being resource sensitive because of its intrinsic ability to reflect computational states, events, and resources [13, 32, 33, 23]. Several notions of game semantics for linear logic are investigated in [6, 1, 2, 15, 19, 17, 9].

Connections between linear logic and probabilistic games considered in complexity theory are investigated in [24, 27, 25]. In particular, linear logic proof search may also be seen as a game. This game, the linear logic proof game, is played on linear logic formulas, and its moves are instances of inference rules of linear logic. There are two players, called proponent and opponent, and a separate verifier. Proponent's goal is to play a sequence of moves that constitute a formal proof of an input formula, consisting of axioms and matching inference rules. Opponent tries to force the direction of proponent's evidence in a way that makes it impossible for proponent to obtain a formal proof. Several versions of this game are discussed in [25, 27], each with a numeric score that reflects the number of certain preferred axioms used in a complete or partial formal proof. The capabilities of the players may differ. While proponent is always omnipotent, in some versions of the game opponent's decisions are based only on a fair coin toss.

Two fragments of propositional linear logic are considered here: the multiplicative additive fragment, **MALL**, and the multiplicative fragment extended with additive constants, **MLLT**. **MALL** is PSPACE-complete [20]. It follows from the NP-completeness of the pure multiplicative fragment, **MLL** [18, 22], that MLL$\top$ is NP-complete. These are *global hardness* properties in that they provide lower bounds on proponent's optimal strategy.

Games from complexity-theoretic literature [5, 14, 28, 34, 11, 8, 7, 16, 30] may be represented in the linear logic proof game, with the new complexity results obtained as corollaries of the complexity properties of games from the literature just mentioned. A representative case is studied here in detail in Section 7. The reader is referred to [25] for an outline of other cases and for a brief overview of the relevant notions and results from complexity theory. The game representations considered in Section 7 are defined in a move-by-move fashion; that is, they preserve proponent's moves, opponent's moves, proponent's strategies, as well as proponent's optimal strategies (that is, optimal with respect to the score).

In this way, one transfers to the linear logic proof game the complexity lower bounds for the approximation of the expected score when proponent plays optimally. In the case of the PSPACE-complete multiplicative-additive fragment of propositional linear logic [20], it is shown in Section 3 that it is as hard to compute an approximation of the optimal score as it is to determine proponent's optimal strategy.

One way to explain this intuitively and informally is that provability in linear logic is not only globally hard, but also *locally hard*. Indeed, in chess and in many other intricate games choosing the best next move often seems just as hard as developing a complete winning strategy. In other words, these games are locally hard. This property is studied in Section 8 for the linear logic proof game. Let us say that an c-heuristic, where $0 < \epsilon < 1$, is a function from formulas to instances of inference rules (that is, proponent's strategy) such that the optimum score arising from the use of this inference rule instance is close (within multiplicative ratio ϵ) of the optimal score. It is shown that unless **P** = **NP**, there is no polynomial-time e-heuristic for MLL$\top$. It is also shown that computing any c-heuristic H for **MALL** would allow us to decide membership in any language in **PSPACE**, using time and space at most a polynomial greater than the time and space needed to compute H.

2 Linear logic proof games

Let p be a propositional atom, let A, B be **MALL** formulas, let $\Gamma, \Delta, \Theta, \Xi$ be finite multisets of **MALL** formulas, and let Σ be a finite multiset of literals or constants 1,O. We write $\Theta \uplus \Xi$ for the (disjoint) multiset union of Θ and Ξ. As usual, we write Γ, A for the multiset obtained by adding an instance of A to Γ. An expression of the form $\vdash \Gamma$ is called a *sequent.* An expression of the form $\vdash \Sigma$ is called a *primitive sequent.*

The English names for **MALL** inference rules are: *identity, cut, par, tensor, bottom, one, plus, with,* and *top.* $\otimes$ and $\parr$ are *multiplicative* connectives; 1 and $\bot$ are *multiplicative* propositional constants. $\oplus$ and $\&$ are *additive* connectives; 0 and $\top$ are *additive* propositional constants. There is no rule for 0. *Linear negation* $^\bot$, mentioned in the identity and cut rules, is defined by recursion on the structure of formulas: $(p^\bot)^\bot$ is p, $(A \otimes B)^\bot$ is $A^\bot \parr B^\bot$, $(A \parr B)^\bot$ is $A^\bot \otimes B^\bot$, $1^\bot$ is I, $\bot^\bot$ is 1, $(A \& B)^\bot$ is $A^\bot \oplus B^\bot$, $(A \oplus B)^\bot$ is $A^\bot \& B^\bot$, $\top^\bot$ is 0, and $0^\bot$ is $\top$.

MALL proof rules are

$$\mathbf{I}\ \frac{}{\vdash p, p^\bot} \qquad \frac{\vdash A, \Gamma \qquad \vdash A^\bot, \Delta}{\vdash \Gamma, \Delta}\ \mathbf{cut}$$

$$\parr\ \frac{\vdash A, B, \Gamma}{\vdash (A \parr B), \Gamma} \qquad \frac{\vdash A, \Gamma \qquad \vdash B, \Delta}{\vdash (A \otimes B), \Gamma, \Delta}\ \otimes$$

$$\bot\ \frac{\vdash \Gamma}{\vdash \bot, \Gamma} \qquad \frac{}{\vdash 1}\ 1$$

$$\oplus\mathbf{1}\ \frac{\vdash A, \Gamma}{\vdash (A \oplus B), \Gamma} \qquad \frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash (A \& B), \Gamma}\ \&$$

$$\oplus\mathbf{2}\ \frac{\vdash B, \Gamma}{\vdash (A \oplus B), \Gamma} \qquad \frac{}{\vdash \top, \Gamma}\ \top$$

MALL enjoys the *cut-elimination property* and the *subformula* **property** [12, 20]. In particular, if a **MALL** formula is provable, then it is provable without the use of the cut rule, and the required proof rules involve only subformulas of the given formula. The fragment MLL$\top$ consists of **MALL** formulas that do not involve $\&$, $\oplus$. The inference rules of **MLLT** are the rules of **MALL** except the rules for $\&$, $\oplus$. The cut-elimination and subformula properties again hold for **MLL** $\top$.

Let us describe several variations of the proof game discussed in [24, 27, 25], all involving the same moves. There are two players, called proponent and opponent, and a separate, *polynomial-time* verifier. Proponent's goal is to play a number of moves demonstrating or giving evidence for a sequent. In order to do this, proponent plays proof rule instances. Opponent tries to force the direction of proponent's evidence in a way that makes it impossible for proponent to win. Opponent plays special markers that may block one side of proponent's $\&$ moves. If proponent plays a $\otimes$ move, then opponent does not block either of the premises. Note that opponent is absent in the case of **MLL** $\top$, that is, the game on **MLL** $\top$ sequents is a kind of solitaire game.

Polynomial-time verifier scores completed plays of the game. Various forms of the game differ in the way they are scored. The main objective of proponent is to never allow opponent to succeed in forcing an unprovable primitive sequent. However, in some forms of the game proponent will be more ambitious, that is, in addition to the main requirement, proponent will try to achieve the best score possible.

Let us first consider a simple version of the game against a randomized opponent, which can be described as an avg/max game played on **MALL** sequents. The game may also be presented as a board game with tiles, where each tile is marked by a linear logic inference rule [24, 26]. Proponent chooses the inference rule to be applied. In the case $\otimes$, proponent chooses a partition and requires both associated expressions to be evaluated. In the case $\oplus$, proponent chooses which of the two expressions will be evaluated. In the case &, opponent chooses by a fair coin toss which of the two expressions will be evaluated. In the case of a primitive sequent, verifier simply computes the value. Each sequent containing the constant $\top$, each identity axiom, and each primitive sequent containing only the constant 1 is scored 1 by verifier. All other primitive sequents are scored 0. Each completed play of the game is scored as the minimum of the scores of terminal sequents obtained in the play. Note that the number of moves is finite; indeed, it is polynomial in the size of a given **MALL** sequent. Proponent wins when each encountered primitive sequent is an identity axiom or the constant 1.

Let us define the function μ, which represents the expected score when proponent plays optimally.,

$$\mu(\Gamma) = max\{\mu(\Gamma'; A) \mid \Gamma = \Gamma', A\},$$

$$\mu(\Gamma; A \otimes B) = max\{min\{\mu(\Delta, A), \mu(\Theta, B)\} \mid \mathbf{A} \uplus \Theta = \mathbf{I}\},$$

$$\mu(\Gamma; A \parr B) = \mu(\Gamma, A, B),$$

$$\mu(\Gamma; A \oplus B) = max\{\mu(\Gamma, A), \mu(\Gamma, B)\},$$

$$\mu(\Gamma; A \& B) = \frac{1}{2}[\mu(\Gamma, A) + \mu(\Gamma, B)],$$

$$\mu(\Gamma; \bot) = \mu(\Gamma),$$

$$\mu(\Gamma; \top) = 1,$$

$$\mu(\Sigma) = \begin{cases} 1 & \text{if } \Sigma \text{ is 1 or an axiom,} \\ 0 & \text{otherwise.} \end{cases}$$

Let us emphasize that, for any **MALL** sequent $\vdash \Xi$, the value $\mu(\Xi)$ is the maximum possible value satisfying these recursive conditions. Specifically, if any encountered sequent contains composite formulas, then several clauses regarding $\mu(\Gamma; A)$ might be applicable. The following proposition is proved by induction on the number of symbols in Ξ.

Proposition 2.1 *A play of the simple linear logic proof game is won by proponent iff the score of the play is equal to* 1. *Furthermore,* **a** **MALL** *sequent* $\vdash \Xi$ *is provable iff* $\mu(\Xi) = 1$. *In addition, if* $\vdash \Xi$ *is unprovuble and does not contain* $\parr$, *then* $\mu(\Xi) = 0$.

However, note that $\mu(\Xi)$ may be arbitrarily close to 1 if $\vdash \Xi$ is unprovable and contains &

The more involved, *weighted* version of the linear logic proof game against a randomized opponent may also be presented as an avg/max game. The players' moves and the winning condition are the same as in the simple game just described. However, in this version of the game, proponent also attempts to use as many certain preferred axioms as possible. Preferred axioms are, say, instances of a distinguished axiom t- $d, d^{\perp}$, where the propositional atom d is fixed in advance. In this version, proponent gets one point for each instance of the distinguished axiom t $d, d^{\perp}$ encountered in a play, but no points are awarded if a primitive sequent $\vdash \Sigma$ is any other identity axiom

Part V

Logic and Programming Languages

Stephen Freund and John Mitchell: "A Type System for Object Initialization in the Java Bytecode Language", to appear in the ACM Transations on Programming Languages and Systems (TOPLAS), ACM Press

Full paper: http://cs.stanford.edu/~freunds/objinit-toplas.ps

Stephen Freund and John Mitchell: "A Formal Framework for the Java Bytecode Language and Verifier", to appear in the Proceedings of the 1999 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications — OOPSLA'99, Denver, CO, November 1999.

Full paper: http: //cs.stanford.edu/~freunds/jvm-type-oopsla.ps

Iliuno Cervesuto: "Logical Framework Design: Why not just classical logic?", accepted for publication in *Information* in *Formation:* Proceedings of the Seventh CSLI Workshop on Logic, Language and Computation (M. Faller, S. Kaufmann, M. Pauly editors), CSLI Publications, 1999.

Full paper:
http://www.stanford.edu/~iliano/papers/Forthcoming/csli99.ps.gz

Stephen N. Freund and John Mitchell: "Specification and Verification of Java Bytecode Subroutines and Exceptions (summary)", Technical Note CS-TN-99-91, Computer Science Department, Stanford University, August 1999

Full paper: http://cs.stanford.edu/"freunds/subexc-tn.ps

Iliano Cervesato, Vuleriu de Paivu and Eike Ritter: "Explicit Substitutions for Linear Logical Frameworks: Preliminary Results", in the Proceedings

of the Workshop on Logical Frameworks and Meta-languages — LFM'99, (A. Felty, editor), Paris, France, 28 September 1999.

Full paper: http://www.stanford.edu/~iliano/papers/lfm99.ps.gz

Stephen N. Freund: "The Costs and Benefits of Java Bytecode Subroutines", in the Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA, 1998.

Full paper: http://theory.stanford.edu/~freunds/subcost.ps

A Type System for Object Initialization In the Java™ Bytecode Language

Stephen N. Freund* John C. Mitchell*
Department' of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu
Phone: (650) 723-2048, Fax: (650) 725-4671

August) 25, 1999

Abstract

In the standard Java implementation, a Java language program is compiled to Java bytecode. This bytecode may be sent across the network to another site, where it is then executed by the Java Virtual Machine. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is run. These checks include type correctness and, as illustrated by previous attacks on the Java Virtual Machine, they are critical for system security. In order to analyze existing bytecode verifiers and to understand the properties that should be verified, we develop a precise specification of *statically-correct* Java bytecode, in the form of a type system. Our focus in this paper is a subset of the bytecode language dealing with object creation and initialization. For this subset, we prove that for every Java bytecode program that satisfies our typing constraints, every object is initialized before it is used. The type system is easily combined with a previous system developed by Stata and Abadi for bytecode subroutines. Our analysis of subroutines and object initialization reveals a previously unpublished bug in the Sun JDK bytecode verifier.

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to a Java bytecode program and this program is then interpreted by the Java Virtual Machine. While many previous programming languages have been implemented using a bytecode interpreter, the Java architecture differs in that programs are commonly transmitted between users across a network in compiled form.

Since bytecode programs may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains *a bytecode verifier* that performs a number of consistency checks

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, and ONR MURI Award N00014-97-1-0505. Stephen Freund received additional support through an NSF Graduate Research Fellowship. A preliminary version of this paper appeared at OOPSLA '98.

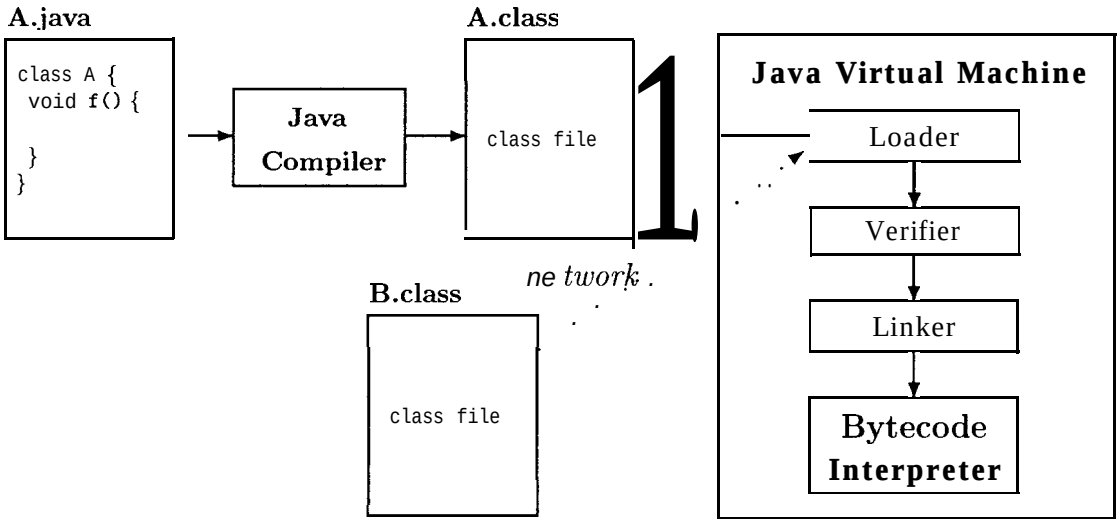


Figure 1: The Java Virtual Machine

before code is executed. Figure 1 shows the point at which the verifier checks a program during the compilation, transmission, and execution process. After a class file containing Java bytecodes is loaded by the Java Virtual Machine, it must pass through thebytecode verifier before being linked into the execution environment and interpreted. This protects the receiver from certain security risks and various forms of attack.

The verifier checks to make sure that every opcode is valid, all jumps lead to legal instructions, methods have structurally correct signatures, and that type constraints are satisfied. Conservative static analysis techniques are used to check these conditions. The need for conservative analysis stems from the undecidability of the halting problem, as well as efficiency considerations. As a result: many programs that would never execute an erroneous instruction are rejected. However, any bytecode program generated by a conventional compiler is accepted. Since mostbytecode programs are the result of compilation. there is very little benefit in developing complex analysis techniques to recognize patterns that could be considered legal but do not occur in compiler output.

The intermediate bytecode language, which we refer to as JVML, is a typed, machine-independent form with some low-level instructions that reflect specific high-level Java source language constructs. For example, classes are a basic notion in JVML, and there is a form of"local subroutine" call and return designed to allow efficient implementation of the source language try-`finally` construct. While some amount of type information is included in JVML to make type checking the possible, there are some high-level properties of Java source code that are not easy to detect in the resulting bytecode program. One example is the last-called first-returned property of the local subroutines. While this property will hold for every JVML program generated by compiling Java source, some effort is required to confirm this property inbytecode programs [SA99].

Another example is the initialization of objects before use. While it is clear from the Java source

language statement

```
A x = new A ((parameters)) ;
```

that the A class constructor will be called before any methods can be invoked through the object reference x, this is not obvious from a simple scan of the resulting JVML program. One reason is that many bytecode instructions may be needed to evaluate the parameters for the call to the constructor. In a bytecode program, these instructions will be executed after space has been allocated for the object and before the object is initialized. Another reason, discussed in more detail in Section 2, is that the structure of the Java Virtual Machine requires copying pointers to uninitialized objects. Therefore, some form of aliasing analysis is needed to make sure that an object is initialized before it is used.

Several published attacks on various implementations of the Java Virtual Machine illustrate the importance of the bytecode verifier for system security. To cite one specific example, a bug in an early version of Sun's bytecode verifier allowed applets to create certain system objects which they should not have been able to create, such as class loaders [DFWB97]. The problem was caused by 'an error in how constructors were verified and resulted in the ability to potentially compromise the security of the entire system. Clearly, problems like this give rise to the need for a correct and formal specification of the bytecode verifier. However! for a variety of reasons, there is no established formal specification; the primary specification is an informal English description that is occasionally at odds with current verifier implementations.

Building on a prior study of the bytecodes for local subroutine call and return [SA99], this paper develops a specification of *statically-correct bytecode* for a fragment of JVML that includes object creation (allocation of memory) and initialization. This specification has the form of a type system, although there are several technical ways in which a type system for low-level code with jumps and type-varying use of stack locations (or registers) differs from conventional high-level type systems. We prove soundness of the type system by a traditional method using operational semantics. It follows from the soundness theorem that any bytecode program that passes the static checks will initialize every object before it is used. We have examined a broad range of alternatives for specifying type systems capable of identifying that kind of error. In some cases, we found it possible to simplify our specification by being more or less conservative than current verifiers. However: we generally resisted the temptation to do so since we hoped to gain some understanding of the strength and limitations of existing verifier implementations. One of these tradeoffs is discussed in Section 6.

In addition to proving soundness for the simple language: we have structured the main lemmas and proofs so that they apply to any additional bytecode instructions that satisfy certain general conditions. This makes it relatively straightforward to combine our analysis with the prior work of Abadi and Stata, showing type soundness for bytecode programs that combine object creation with subroutines. In analyzing the interaction between object creation and subroutines, we have identified a previously unpublished bug in the Sun implementation of the bytecode verifier. This bug allows a program to use an object before it has been initialized; details appear in Section 7. Our type-based framework also made it possible to evaluate various ways to fix this error and prove correctness for a modified system.

Section 2 describes the problem of object initialization in more detail, and Section 3 presents JVML,, the language which we formally study in this paper. The operational semantics and type system for this language is presented in Section 4. Some sound extensions to our type system, including subroutines, are discussed in Section 6, and Section 7 describes how this work relates to Sun's implementation. Section 8 discusses some other projects dealing with bytecode verification, and Section 9 gives directions for future work and concludes.

2 Object Initialization

As in many other object-oriented languages, the Java implementation creates new objects in two steps. The first step is to allocate space for the object. This usually requires someenvironment-specific operation to obtain an appropriate region of memory. In the second step, user-defined code is executed to initialize the object. In Java, the initialization code is provided by a constructor defined in the class of the object. Only after both of these steps are completed can a method be invoked on an object.

In the Java source language, allocation and initialization are combined into a single statement, as illustrated in the following code fragment:

```
Point p = new Point (3) ;
p.Print () ;
```

The first line indicates that a new Point object should be created and calls the Point constructor to initialize this object. The second line invokes a method on this object and, therefore, can be allowed only if the object has been initialized. Since every Java object is created by a statement like the one in the first line here, it does not seem difficult to prevent Java source language programs from invoking methods on objects that have not been initialized. While there are a few subtle situations to consider, such as when a constructor throws an exception, the issue is essentially clear cut.

It is much more difficult to recognize initialization-before-use in bytecode. This can be seen by looking at the five lines of bytecode that are produced by compiling the preceding two lines of source code:

```
1: new #1 <Class Point>
2:   dup
3:   iconst_3
4:   invokespecial #4 <Method Point(int)>
5:   invokevirtual #5 <Method void Print () >
```

The most striking difference is that memory allocation (line 1) is separated from the constructor invocation (line 4) by two lines of code. The first intervening line, dup, duplicates the pointer to the uninitialized object. The reason for this instruction is that a pointer to the object must be passed to the constructor. As a convention of the stack-based virtual machine architecture, parameters to a function are popped off the stack before the function returns. Therefore, if the address were not duplicated, there would be no way for the code creating the object to access it after it is initialized. The second line, iconst-3 pushes the constructor argument 3 onto the stack. If p were used again after line 5 of the bytecode program, another dup would have been needed prior to line 5.

Depending on the number and type of constructor arguments, many different instruction sequences may appear between object allocation and initialization. For example, suppose that several new objects are passed as arguments to a constructor. In this case, it is necessary to create each of the argument objects and initialize them before passing them to the constructor. In general, the code fragment between allocation and initialization may involve substantial computation, including allocation of new objects, duplication of object pointers, and jumps to or branches from other locations in the code.

Since pointers may be duplicated, some form of aliasing analysis must be used. More specifically, when a constructor is called, there may be several pointers to the object that is initialized as a result, as well as pointers to other uninitialized objects. In order to verify code that uses pointers to initialized objects, it is therefore necessary to keep track of which pointers are aliases (name the same object). Some hint for this is given by the followingbytecode sequence:

A Formal Framework for the Java Bytecode Language and Verifier

Stephen N. Freund* John C. Mitchell*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu
Phone: (650) 723-8634, Fax: (650) 725-4671

Abstract

This paper presents a sound type system for a large subset of the Java bytecode language including classes, interfaces. constructors. methods, exceptions, and bytecode subroutines. This work serves as the foundation for developing a formal specification of thebytecode language and the Java Virtual Machine's bytecode verifier. We also describe a prototype implementation of a type checker for our system and discuss some of the other applications of this work. For example, we show how to extend our work to examine other program properties, such as the correct use of object locks.

1 Introduction

The bytecode language, which we refer to as JVML, is the platform independent representation of compiled Java programs. In order to prevent devious applets from causing security problems stemming from type errors! the Java Virtual Machine bytecode verifier performs a number of consistency checks on bytecode before it is executed [LY96]. This paper presents a type system that may serve as the foundation of a formal specification of the bytecode verifier for large fragment of JVML. Originally, the only specification was an informal English description that was incomplete or incorrect in some respects. Since then, a variety of more formal specifications for various subsets of JVML have been proposed. We discuss some of these in Section 9.

In our previous studies, we have examined several of the complex features of JVML in isolation. One study focused on object initialization and formalized the way in which a type system may prevent Javabytecode programs from using objects before they have been initialized [FM98]. In other work, we extended the work of Stata and Abadi on bytecode subroutines to develop a semantics for subroutines that is closer to the original Sun specification and includes exception handlers [FM99]. Subroutines are a form of local call and return that allow for space efficient compilation of try-f `inally` structures in the Java language. Bytecode programs that use subroutines are allowed to manipulate return addresses in certain ways, and the bytecode verifier must ensure that these return addresses are used appropriately. In addition, subroutines introduce a limited form of polymorphism into the type system.

This paper builds on these studies to construct a formal model for a subset of JVML containing:

- classes, interfaces; and objects
- constructors and object initialization
- virtual and interface method invocation
- arrays
- exceptions and subroutines
- integer and double primitive types

This subset captures the most difficult static analysis problems in bytecode verification. There are some features of JVML that we have not covered, including additional primitive types and control structures, final and static modifiers, access levels, concurrency, and packages. In some sense, these omitted features only contribute to the complexity of our system in the sheer number of cases that they introduce, but they do not appear to introduce any challenging or new problems. For example, the excluded primitive types and operations on them are all very similar to cases in our study, as are the missing control structures, such as the tableswitch instruction. Static methods share much in common with normal methods, and checks for proper

*Supported in part by NSF grants CCR-9303099 and CCR-9629754. and ONR MURI Award N00014-97-1-0505. Stephen Freund received additional support through a NSF Graduate Research Fellowship.

To appear in OOPSLA '99

use of final and access modifiers are well understood and straightforward to include.

We have applied our previous work on developing type checkers for the JVML type system to cover the subset presented in this paper as well [FM99] and have implemented a prototype verifier. This prototype demonstrates that our type system does reject faulty programs and also accepts virtually all programs generated by reasonable Java compilers.

Although the main contribution of this work is a framework in which bytecode verification can be formally specified, our type system is also useful for other purposes. We have extended our system to check additional safety properties, such as ensuring that object locks are acquired and released properly by each method. In addition, augmenting the verifier to track more information through the type system has allowed us to determine where run-time checks, such as null pointer and array bounds tests: may be eliminated.

This work may also lead to methods for verifying bytecode programs offline in situations where full bytecode verification cannot be done in the virtual machine due to resource constraints.

Section 2 introduces JVML_f, the fragment of JVML studied in this paper. Sections 3, 4 and 5 describes the formal dynamic and static semantics for JVML_f and gives an overview of the soundness proof, and Section 6 highlights some of the technical details in handling object construction and subroutines. Section 7 gives a brief overview of our prototype verifier. Section 8 describes some applications of this work, Section 9 discusses some related work, and Section 10 concludes.

2 JVML_f

In this section, we informally introduce JVML_f, an idealized subset of JVML encompassing the features listed above. We shall use the Java program in Figure 1 as an example throughout the section. Compilation of this code produces a class file for each declared class or interface. In addition to containing the bytecode instructions for each method, a class file also contains the symbolic name and type information for any class, interface! method, or field mentioned in the source program. This information allows the Java Virtual Machine to verify and dynamically link code safely. To avoid some unnecessary details inherent in the class file format, we shall represent JVML_f programs as a series of declarations somewhat similar to Java source code, as demonstrated in Figure 2.

The collection of declarations in Figure 2 contains the compiled Java language classes Object and Throwable. These are included in the JVML_f program so that all referenced classes are present. As a convenience, we assume that these are the only two library classes and that they are present in all JVML_f programs. The JVML_f declaration for each class contains the set of instance fields for objects of that class, the interfaces declared to be implemented by the class, and all the methods declared in the class. Each method consists of an array of instructions and a list of exception handlers, and all methods in the superclass of a class are copied into the subclass unless they are overridden. Although real class files do not duplicate the code in this manner, it simplifies several aspects of our system by essentially flattening the class hierarchy when it comes to method lookup. The special name `<init>` is given to constructors.

The execution environment for JVML_f programs consists of a stack of activation records and an object store. Each activation record contains a program counter, a local operand stack, and a set of local variables. These pieces of information are not shared between different activation records, although different activation records may contain references to the same objects in the heap. Most JVML_f bytecode instructions operate on the operand stack, and the store and load instructions are used to store intermediate values in the local variables. Constructing or deleting activation records upon method invocation or return is left to the Java Virtual Machine.

Figure 3 contains the full JVML_f instruction set. and the next few paragraphs briefly describe the interesting aspects of these instructions. In Figure 3, v is an integer, real number, or the special value `null`; x is a local variable; L is an instruction address; and σ and τ are a class name and array component type, respectively. We refer the reader to the Java Virtual Machine specification for a detailed discussion of these bytecode instructions [LY96].

The bytecode language uses *descriptors* to refer to method and field names from Java language programs. A descriptor contains three pieces of information about the method or field that it describes:

- the class or interface in which it was declared
- the field or method name
- its type

For example, the bytecode instruction used to set the num instance field in the constructor for A is putf ield $\{|\mathtt{A}$, num, $\mathtt{int}|\}_{\mathtt{F}}$. Descriptors are used in place of simple names to provide enough information to:

1. check uses of methods and fields without loading the class to which they belong.
2. dynamically link class files safely.

```
interface Foo {
  int foo(int y);
}

class A extends Object implements Foo {
  int num;
  A(int x) {
    num = x;
  }

  int foo(int y) {
    A a;
    try {
      a = new A(y);
    } catch (Throwable e) {
      num = 2;
    }
    return 6;
  }
}
```

```
class B extends A {
  A array [];
  B(int x) {
    super(x);
    num = foo(2);
  }
}
```

Figure 1: Declaration of several Java classes.

```
class Object {
  super: None
  fields:{}
  interfaces: {}
  methods:
}
```

```
class Throwable {
  super: Object
  fields:{}
  interfaces: {}
  methods:
}
```

```
interface Foo {
  interfaces: {}
  methods:{{|Foo, foo, int → int|}_I}
}
```

```
class A {
  super: Object
  fields: {{|A, num, int|}_F}
  interfaces: {Foo}
  methods:
    {|A, <init>, int → void|}_M {
      1:  load 0
      2:  invokespecial {Object, <init>, ϵ → void|}_M
      3:  load 0
      4:  push 1
      5:  putfield {|A, num, int|}_F
      6:  return
    }
    {|A, foo, int → int|}_M {
      1:  new A
      2:  store 2
      3:  load 2
      4:  load 1
      5:  invokespecial {|A, <init>, int → void|}_M
      6:  goto 12
      7:  pop
      8:  load 0
      9:  push 2
     10:  putfield {|A, num, int|}_F
     11:  goto 12
     12:  push 6
     13:  returnval
     Exception table:
        from   to target  type
          1     6    7     Throwable
    }
}
```

```
class B {
  super: A
  fields: ({/A, num, int|}_F,
           {|B, array, (Array A)|}_F}
  interfaces: {Foo}
  methods:
    {|B, <init>, int → void|}_M I
      1:  load 0
      2:  load 1
      3:  invokespecial {|A, <init>, int → void|}_M
      4:  load 0
      5:  load 0
      6:  push 2
      7:  invokevirtual {|A, foo, int → int|}_M
      8:  putfield {|A, num, int|}_F
      9:  return
    }
    {|B, foo, int → int|}_M {
      /* as in superclass */
    }
}
```

Figure 2: Translation of the code from Figure 1 into JVML_f.

```
instruction ::=  push v | pop | store x | load x
               | add | ifeq L | goto L
               | new σ
               | invokevirtual MDescriptor
               | invokeinterface IDescriptor
               | invokespecial MDescriptor
               | getfield FDescriptor
               | putfield FDescriptor
               | newarray (Array τ) | arraylength | arrayload | arraystore
               | throw σ | jsr L | ret x
               | return | returnvalue
```

Figure 3: The JVML_f instruction set

3. provide unique symbolic names to overloaded methods and fields. Overloading is resolved at compile time in Java.

Valid method, interface: and field descriptors are generated by the following grammar:

$$\begin{aligned} \textit{MDescriptor} &::= \{\textit{Class-Name. Label. Method-Type}\}_{\text{M}} \\ \textit{IDescriptor} &::= \{\textit{Interface-Name, Label, Method-Type}\}_{\text{I}} \\ \textit{FDescriptor} &::= \{\textit{Class-Name, Label. Field-Type}\}_{\text{F}} \end{aligned}$$

A *Field-Type* may be either int. float, any class or interface name, oan array type. A*Method-Type* is a type a $\rightarrow \gamma$ were α is a possibly empty sequence of *Field- Type*'s and γ is the return type of the function (or void). Figure 4 shows the exact representation of all of these types. as well as several additional types and type constructors used in the static semantics but not by any JVML_f program. For example. the type Top, su-e pertype of all types. will be used in the typing rules. but cannot be mentioned in JVML_f program. Note that we distinguish between methods declared in a class and methods declared in an interface using different types of descriptors.

Some JVML_f instructions generate exceptions when the arguments are not valid. For example, if null is used as an argument to any instruction that performs an operation on an object, a run-time exception is generated. The value of an exception is an object whose class is Throwable or a subclass of it. To avoid introducing additional classes, we assume that all failed run-time checks generate Throwable objects. When an exception is generated. the list of handlers associated with the currently executing method is searched for an appropriate handler. An appropriate handler is one declared to protect the current instruction and to handle exceptions of the class of the object that was thrown or some superclass of it. If an appropriate handler is found, execution jumps to the first instruction of the exception handler's code. Otherwise, the top activation record is popped, and this process is repeated on the new topmost activation record.

3 Dynamic Semantics

This section gives an overview of the formal execution model for JVML_f programs. Section 3.1 describes the representation of programs as an environment, Section 3.2 introduces a few notational conventions, and Section 3.3 describes the semantics of bytecode instructions.

3.1 Environments

A JVML_f program is represented formally by an environment I' containing all the information about classes. interfaces, and methods found in the class files of the program. The environment, as defined in Figure 5, is broken into three components storing each of these items. Construction Γffrom the representation of class file information demonstrated in Figure 2 is straightforward.

We write$\Gamma \vdash \tau_1 <: \tau_2$ to indicate thatτ_1 is a subtype ofτ_2, givenΓ. The Java Virtual Machine model uses this judgment as a way to perform run-time type tests. The subtyping rules are presented in the Appendix, and they follow the form of the rules used to model subtyping in the Java langu[DE97, Sym97], with extensions to cover tJVML_f specific types. This judgment, and all others presented in this paper, are summarized in Figure 6.

3.2 Notational Conventions

Before proceeding, we summarize a few notational conventions used throughout this paper. To access information about a metho$M$ declared inΓ, we write$\Gamma[M]$. Similar notation is used to access interface and class declarations. I$\Gamma[M] = \langle P, \text{H}\rangle$ for some method descriptor$M$, then $Dom(P)$ is the set of addresses from the set ADDR used in$P$, and $P[i]$ is the i^{th} instruction in$P$. $Dom(P)$ will always include address 1 and is usually a rang$\{1, \ldots, n\}$ for some$n$. Likewise,H is a partial map from integer indexes to handlers, where

Logical Frameworks

Why not just classical logic?

Iliano Cervesato

I was recently invited to give a presentation about the logical framework *LLF*. After a 40 minutes talk in which I revealed the intricacies of the underlying type theory and illustrated by means of examples the meta-representation wonders of this new language, somebody in the audience said: "This looks very complicated. Why not using, say, classical logic instead?". In this chapter, I build upon my then improvised answer. I will recall what logical frameworks are and try to motivate the simple but unfamiliar constructs they often rely upon.

1 Introduction

It is often taught in introductory classes in Philosophy, Mathematics, and Computer Science that logic is the universal language of reasoning and rigorous representation of knowledge. This is not unfounded: for example, the entire body of mathematics can be formalized in classical first-order predicate logic.

This however causes people with only a basic logical background to frown when they hear logical framework experts, those scholars who specialize in formal reasoning and knowledge representation, use scary-sounding words such as "higher-order abstract syntax", "dependent type theories", and "linearity". If classical logic is so universal, why do these authors rebuff it for those apparently cryptic and hopelessly complicated languages?

In this chapter, we show that many ideas in modern logical frameworks emerge as refinements of computationally or representationally suboptimal aspects of classical logic. Our progression does not reflect the historical development of these sys-

Proceedings of the Seventh CSLI Workshop on Logic, Language and Computation.
Martina Failer, Stefan Kaufmann & Marc Pauly (eds.)

terns, and is geared towards logical frameworks in the *LF* family [Harper 1993, Pfenning 1991].

2 Deductive Systems

Exchanging information and reasoning about a formalism, may it be a novel programming language or an arcane algebraic structure, presupposes that we have a language to express the concepts we are interested in. These include basic linguistic facts such as "3 + 5 is a well-formed expression" *(syntax)*, descriptions of the effect of operations, for example "3 + 5 evaluates to 8" *(semantics)*, and properties of the formalism such as "+ is commutative" *(meta-theory)*. In these three examples, we used English to express the concepts involved. Being fully rigorous when using natural languages is difficult to achieve and impossible to enforce. Jargon and abbreviations (e.g. writing "3 + 5 $\hookrightarrow$ 8" for "3 + 5 evaluates to 8") only alleviate the problem.

Deductive systems [Martin-Löf 1985] approach rigor by fixing precise conventions on how to express the concepts of the formalism we are interested in (the object language). In those presentations, the predications we make about object entities are called *judgments*. The quoted examples above are all judgments.

In their simplest instance, judgments are syntactic descriptions of relations among lexical elements of the object language. They can either *hold (e.g. "3 + 5 $\hookrightarrow$ 8")* or *fail to hold (e.g.* "3 + 5 $\hookrightarrow$ 9"). Expressing the meaning of the constructs of the object language amounts then to specifying which of the relevant judgments hold, and which do not. An explicit and exhaustive enumeration of the former (and of the latter) is generally ineffective since infinitely many judgments may be involved ($\_ + \_ \hookrightarrow \_$ is an example). However, the constructs commonly found in all formalisms of interest display forms of regularity that make them amenable to a finite description. Schematic *rules of* **inference** achieve this effect by expressing the validity of classes of judgments sharing a common syntactic pattern (rule conclusion) in terms of the validity of zero or more other judgments *(rule premises)*. Instances of inference rules are chained to provide ev-

idence of the validity of specific judgments in the form of finite *derivations*. A judgment is then derivable if it has a derivation. A deductive system faithfully expresses an object formalism if all and only the judgments that hold are derivable. We will now illustrate these definitions on our running example.

Notations for grammars such as the Backus-Naur Form yield deductive systems for the syntax of a formalism. If, for the sake of succinctness, we express natural numbers in unary notation, we can specify the syntax of our example as follows:

$$num ::= z \mid s\ num$$
$$exp ::= num + num$$

where we have chosen the symbols z, s, and + to denote zero, successor, and addition, respectively. The implicit judgments here are "the string _ is a number" and similarly for expressions. The inference rules are the grammatical productions: for example, the first line states that "z is a number" (rule with no premises), and that "s N is a number" if "N is a number" (rule schematic in N with one premise). The derivations are all the parse trees.

Here, "s" can be viewed either as a character (or token) so that "s s z" is a list of characters (tokens), or as a function symbol with one arguments so that "s s z" stands for the expression "s (s (z))". The former approach is called *concrete syntax*, the latter *abstract syntax*. Expressions in the abstract syntax are isomorphic to parse-trees.

Given this representation of numbers, a simple recursive definition specifies how to evaluate their sum: "adding zero to any number N yields N", and "for any numbers M, N, and V, s M + N evaluates to s V if M + N evaluates to V". Transliterating each part into symbols yields the following two rules:

$$\frac{N\ num}{z + N \hookrightarrow N}\ \textbf{eval\_z} \qquad \frac{N, M, V\ num \qquad M + N \hookrightarrow V}{s\,M + N \hookrightarrow s\,V}\ \textbf{eval\_s}$$

The horizontal line separates the premises and the conclusion of each rule, and the text on the right identifies the rule. The first rule has one premise and is schematic in N, the second has four (abbreviated) premises and is schematic in M, N, and V. It

is easy to observe that the syntactic judgments in rule **evals** are redundant, but omitting the premise of rule **eval_z** would allow successfully evaluating garbled expressions (e.g. $\mathrm{z}+oops \hookrightarrow$ oops). Observe that if we think of each rule as describing an atomic step of the evaluation of the sum of two numbers, then derivations are a notation for evaluation traces.

Meta-theoretic properties are predications over semantic derivations. For example, "+ is commutative" can be restated as "given numbers M, N, and V, for every derivation $\mathcal{D}$ of $M+N \hookrightarrow V$, there exists a derivation $\mathcal{D}'$ of $N + M \hookrightarrow V$". Therefore, a convenient notation for derivations is an essential prerequisite for reasoning about a formalism. For space reasons, we refrain from further discussing meta-theoretic judgments (properties) and their derivations (proofs). The techniques we will illustrate are however applicable also in that setting (see [Michaylov 19911).

3 Logical Frameworks

Through the notions of judgment and derivation, deductive systems allow precise descriptions of formalisms in Mathematics, Logic: and Computer Science. However, when exchanging ideas with others or proving properties, we seldom adhere to their full formality: their rigid patterns soon get in the way of effective communication. A variously balanced mixture of natural language, judgments, and derivation sketches is normally adopted as a good compromise between rigor and bearability.

Formalizing even simple proofs often requires a fair amount of work with little benefit: indeed, the formal argument is seldom more convincing than the original proof since we, as humans, have a limited ability of keeping alert when confronted with long and convoluted chains of inference. Paradoxically, formal errors are more likely to pass unnoticed than informal ones.

Since computers are free from the attention shortcomings of the human brain, they are ideal candidates for the clerical work of checking proofs and derivations, and, in simple cases, of validating judgments. Parsers, interpreters, compilers, and various related tools efficiently mechanizes aspects of the syntax, seman-

Specification and Verification of Java Bytecode Subroutines and Exceptions (summary)

Stephen N. Freund* John C. Mitchell*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu
Phone: (650) 723-8634, Fax: (650) 725-4671

August, 25, 1999

Abstract

This paper develops a formal description of verifying Java bytecode subroutines and exceptions. We first present a type system for subroutines which extends the work of Stata and Abadi to encompass multilevel returns [SA98]. Second, we describe a data flow analysis algorithm proven to accept exactly the set of well-typed programs. Finally, we extend our type system and algorithm to include exception handlers and exceptions. We also give a brief overview of an implementation of our type checker.

1 Introduction

Execution of mobile Java code can potentially lead to significant security problems if the code is not carefully monitored to prevent it from accessing sensitive data, deleting files, or corrupting the runtime system in other ways. The Java Virtual Machine has security mechanisms in place to prevent these kinds of attacks [Gon98], but they can be circumvented by a devious applet if it can cause certain run-time type errors, such as casting an integer to a pointer, as demonstrated in [DFW96, Sar97].

The Java Virtual Machine *bytecode* verifier prevents these types of errors by performing a number of consistency checks on compiled programs before they are executed. Compiled programs are written in what we shall call JVML, a platform-independent bytecode language with some low-level instructions to reflect high-level features and constructs of the Java language, such as exception handlers, classes, etc. The verifier checks for these programs include determining that:

- every instruction opcode is valid.
- all jumps are to valid instruction addresses.

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, ONR MURI Award N00014-97-1-0505. Stephen Freund received additional support through a NSF Graduate Research Fellowship.

```
void f() {
  try {
    something();
  } catch(Exception e) {
    oopso;
  } finally {
    done0;
  }
}
```

Figure 1: A method using a try-finally statement.

- operations are not performed on values of incorrect type.

This work studies the most complex parts of JVML and develops a foundation for formal specification of the verifier in terms of a type system and type checking algorithm.

We first present a new type system for JVML_0, a small fragment of JVML encompassing some simple operations and subroutines [SA98]. Bytecode subroutines are a form of local call and return that provide space efficient compilation of the try-finally statements in Java programs, as demonstrated in Figures 1 and 2. Without subroutines, the code for the finally block would have to be duplicated at every exit from the try block. However, verification of programs using subroutines is complicated by the need to check that

1. subroutine calls and returns occur in a stack-like manner, with the exception of multilevel returns, which are discussed below.

2. a specific form of local variable polymorphism introduced by subroutines is used correctly. In particular, local variables not touched by a subroutine may contain values of conflicting types at different calls to the subroutine.

We use some of the ideas from the type system of Hagiya and Tozawa [HT98] to eliminate one of the major simplifications to the subroutine mechanism made by Stata and Abadi and to develop a type system based on their work that is closer to the original Sun specification. In comparison to Stata and Abadi, our system is able to type check multilevel returns, which were not allowed in [SA98].

A multilevel return occurs when a subroutine returns not to its caller, but to its caller's caller, or even further up the stack of subroutines which have been called but which have not exited. As described in Section 9, our JVML_0 type system differs technically from Hagiya and Tozawa and the wide variety of other type systems for JVML_0 (see, for example, [HT98, O'C99, Qia98]), and our system offers some advantages over the others. Most importantly, we feel that ours is the closest to the Sun specification and existing implementations.

We also present a verification algorithm proven to accept exactly the set of well-typed programs. The algorithm uses standard data flow analysis techniques to synthesize type information for a program. However, subroutines again complicate matters. Our verifier computes flow control and variable usage information about subroutines before beginning data flow analysis. Precomputation of this information simplifies both the implementation and correctness proofs for the data flow analysis aspect of the type checker in contrast to some of the other suggested type checking algorithm [HT98, Qia99] that compute all information at once.

```
Method void f ()
// code for try block
 0 aload_0                                  // load this
 1 invokevirtual <Method void something()>  // call something
 4 jsr 23                                   // call subroutine
 7 return

// code for exception handler
 8 pop                                      // ignore exception value
 9 aload_0                                  // load this
10 invokevirtual <Method void oops()>       // call oops
13 jsr 23                                   // call subroutine
16 return

// compiler-inserted handler
17 astore_1                                 // store exception value to rethrow
18 jsr 23                                   // call subroutine
21 aload_1                                  // load exception
22 athrow                                   // rethrow

// subroutine
23 astore_2                                 // store return address
24 aload_0                                  // load this
25 invokevirtual <Method void done()>       // call done()
28 ret 2                                    // return from subroutine

Exception table:
   from   to target type
     0     4     8   <Class java.lang.Exception>
     0    13    17   any
```

Figure 2: The translation of the program in Figure 1 into JVML. Control is transfered to the subroutine beginning on line 23 at the end of the try block, at the end of the exception handler, and when an uncaught exception (handled at line 17) causes an abrupt exit from the method.

The second half of this paper presents a type system and verifier for JVML,, an extension of JVMLa containing exceptions and exception handlers. The machinery developed to check multilevel returns may also be applied to type check exception handlers in the presence of subroutines. A jump in execution caused by throwing and catching an exception may, in some cases, behave very similar to a multilevel return. This is the first treatment of exception handlers and subroutines in a framework derived from the work of Stata and Abadi and the first presentation of a provably correct type checking algorithm similar to the original Sun specification.

We also summarize our experience with prototype implementations of the type checking algorithms.

Our type system and verification algorithm for JVML, serves as the core of a specification and implementation of a bytecode verifier for all of JVML. We have extended this work to include our previous work on object initialization [FM98], and we have in fact been able to construct a sound type system for a much larger fragment of JVML that includes classes, interfaces: methods, constructors, subroutines, and exceptions based on these smaller systems.

Sections 2, 3, and 4 present JVMLa, its formal semantics, and a type checker for it. We extend our type system and verifier to include exception handlers in Sections 5 and 6. Section 7 describes an implementation of our algorithms, and Section 8 compares our verifier to the Sun implementation. Section 9 relates our contributions to other work on the bytecode verifier, and Section 10 draws some conclusions about this project.

2 JVML_0

A JVML_0 program is an array P of instructions drawn from the following list, where the basic execution model consists of a program counter, a set of local variables, and an operand stack:

$$
\begin{aligned}
\mathit{instruction} ::= \ & \mathtt{push\ 0} \mid \mathtt{inc} \mid \mathtt{pop} \\
& \mid \mathtt{if}\ L \\
& \mid \mathtt{store}\ x \mid \mathtt{load}\ x \\
& \mid \mathtt{jsr}\ L \mid \mathtt{ret}\ x \\
& \mid \mathtt{halt}
\end{aligned}
$$

where x is a local variable name and $L \in \textsc{Addr}$, the set of instruction addresses. Informally, push 0, inc, pop, and if L perform the obvious operations on the stack; store x and load x move values between the static and local variables; jsr L jumps to instruction L, pushing the return address onto the stack; and ret x jumps to the instruction address stored in local variable x.

3 Semantics of JVML_0

This section describes the dynamic and static semantics for JVMLa

3.1 Dynamic Semantics

The bytecode interpreter for JVMLa is modeled using the standard framework of operational semantics. The rules characterize transitions between machine states of the form $\langle pc, f, \mathrm{s}\rangle$ where

- pc is a program counter
- f is a total map from $\textsc{Var}$, the set of local variables, to values.

Explicit Substitutions for Linear Logical Frameworks: *Preliminary Results* *

Iliano Cervesato
Computer Science Department
Stanford University
Stanford, CA 94305-9045 — USA
iliano@cs.stanford.edu

Valeria de Paiva and Eike Ritter
School of Computer Science
University of Birmingham
Birmingham, B15 2TT — UK
{*V.DePaiva*|*E.Ritter*}*@cs.bham.ac.uk*

Abstract

We present the calculus xdLLF$^-$ and experiment with aspects of its meta-theory. xdLLF$^-$ integrates linear explicit substitutions in de Bruijn notation into the simply-typed fragment of the linear logical framework LLF. After observing that the expected σ-rules invalidate subject reduction, we devise a specification of σ-normalization inspired by the big-step semantics of programming languages, and prove it correct.

1 Introduction

Explicit substitutions [1] have been used to rationalize the implementation of many systems based on various X-calculi, such as functional languages, logical frameworks, and higher-order logic programming languages. As linear X-calculi have grown in popularity, so has the need for solid and efficient support for their implementation. A linear adaptation of explicit substitution techniques is a prime candidate. The authors of this paper have separately explored this possibility in two distinct settings:

- In [6], Ghani, de Paiva, and Ritter have designed the language xDILL, geared towards the implementation of functional languages. It is based on Barber and Plotkin's DILL (Dual Intuitionistic Linear Logic) [2], and is characterized, among other things, by variables of two different kinds: linear variables are used exactly once, and intuitionistic variables can be accessed arbitrarily many times. The extra information about usage of linear variables makes it possible to apply various optimizations like update-in-place of aggregate data structures such as arrays, or savings in memory allocation. This significantly influenced the design decisions of the calculus in [6].
- On the other hand, Cervesato and Pfenning have based their implementation of the linear logical framework LLF [4] on a form of linear explicit substitution, although they did not thoroughly investigate its meta-theory. LLF is a close relative of DILL (for example, both distinguish linear and intuitionistic variables). LLF is however designed as a logical framework, which forces a set of operations on terms that are not found in DILL. An implementation of LLF must support

*The first author was partially supported by DoD MURI, "Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505. The second and third authors were partially funded by ESPSRC grant GR/28296, "The eXplicit Substitution Linear Abstract Machine".

term reconstruction to make meta-representation practical, permit logic programming-style proof-search, and accommodate the forthcoming addition of theorem proving capabilities. Each of these functionalities relies on (higher-order) unification and therefore an explicit substitution calculus for LLF must handle meta-variables and their manipulation.

In this paper, we bring our experiences together in trying to isolate some of the issues that arise when combining linearity and explicit substitutions in our different settings. Although the results reported here are very preliminary, this work had the effect of furthering our understanding of these problems.

We start from the linear X-calculus LLF^- (Section 2), which includes operators from both LLF and DILL while ignoring complex features such as dependent types and an unrestricted "!" operator. LLF^- enjoys properties such as subject reduction, normalization and confluence. Then, by using a standard process [1], we construct the calculus xdLLF^- (Section 3) which incorporates substitutions as a separate syntactic category in LLF^- (along the way, we also switch to a de Bruijn notation, motivated by our interests in efficient implementations). This has the positive effect of turning the meta-level substitutions produced by the P-reductions into explicit substitutions that can be manipulated within the calculus. At this point, the standard approach [1] would require us to express the implicit procedure to carry out the application of a meta-level substitution as a set of rewrite rules about explicit substitutions (g-rules): the transcription is correct if we can prove that there is a reduction strategy which eliminates all explicit substitutions and terminates with the X-term that would be produced by making all explicit substitutions implicit.

We deviate from this path since the o-rules we would obtain for xdLLF^- interfere with linearity and allow rewriting well-typed terms into ill-typed objects. In [6], we solved this problem by splitting the linear substitutions according to the usage of the linear variables. This approach may cause a significant overhead when implemented, and does not scale up when extending xdLLF^- with metavariables. We instead explore a different path (Section 4): we give a syntactic characterization of the set of o-normal terms (to which no c-rule would be applicable) as the language xdLLF^-_σ, outline a type-preserving procedure that reduces a typable xdLLF^- term to its g-normal form, and prove its correctness. Although this approach deals correctly with linearity, it still has several drawbacks. First, it fixes the reduction strategy, which is instead open when a-rules are used. Second, it does not allow interleaving σ-normalization steps with other reductions. Third, it does not scale up to handle meta-variables. Nonetheless, we see it as a valuable first step toward addressing these issues more satisfactorily (Section 5).

2 LLF^-

The calculus LLF^-, that we use as our starting point, enriches the simply-typed fragment of the language of the linear logical framework LLF [4] with multiplicative pairs and unit. On the other hand, it extends the language DILL [2, 6] with additives and with intuitionistic functions, but sacrifices its full-fledged exponential "!". LLF^- is defined as follows:

Types: $A ::=$		*Terms:* $M ::= x$			
	$\top$		0		(additive *unit)*
	$A_1 \,\&\, A_2$		$\langle M_1, M_2\rangle$	$\mathsf{fst}\, M \mid \mathsf{snd}\, M$	(additive pairs)
	$\mathbf{1}$		$\bullet$	$\mathsf{let}\, M_1\, \mathsf{be} \bullet \mathsf{in}\, M_2$	(multiplicative unit)
	$A_1 \otimes A_2$		$M_1 \otimes M_2$	$\mathsf{let}\, M_1\, \mathsf{be}\, x_1 \otimes x_2\, \mathsf{in}\, M_2$	(multiplicative pairs)
	$A_1 \multimap A_2$		$\hat{\lambda} x \hat{:} A.\, M$	$M_1 \hat{} M_2$	(linear functions)
	$A_1 \to A_2$		$\lambda x : A.\, M$	$M_1\, M_2$	(intuitionistic functions)

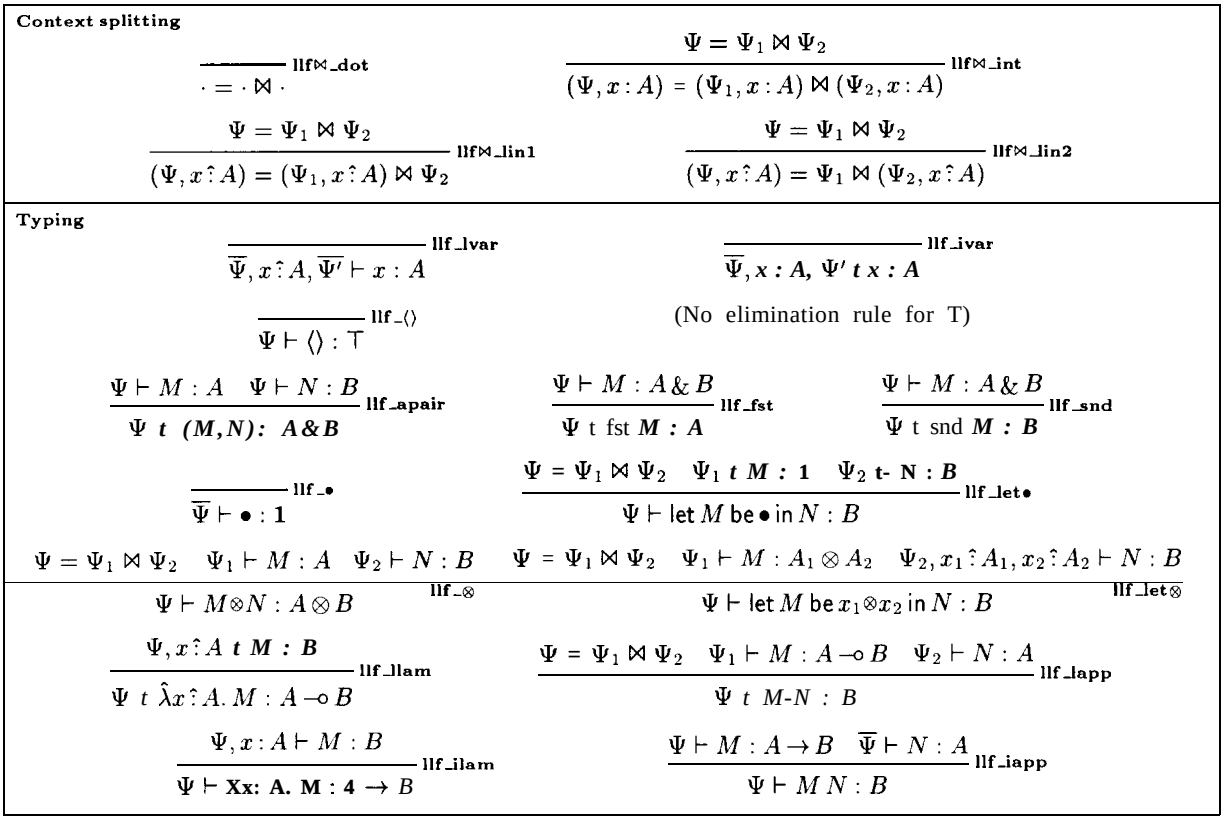


Figure 1: Typing in LLF^{-}

Here x and *a* range over variables and base types, respectively. In addition to the names displayed above, we will often use N and B for terms and types, respectively. As usual, we rely on a context to assign types to free variables.

$$\textit{Contexts:}\quad \Psi ::= \cdot \mid \Psi, x:A \mid \Psi, x \hat{:} A$$

where $x \hat{:} A$ and x : *A* stand for a linear and a reusable (intuitionistic) assumption of type *A*, respectively.

The notions of free and bound variables are adapted from the simply typed X-calculus. As usual, we identify terms that differ only by the name of their bound variables. Contexts are treated as sequences, we promote "," to denote their concatenation and omit writing "·" when unnecessary. As usual, we require variables to be declared at most once in a context. Finally, we write $\overline{\Psi}$ for the intuitionistic part of context Ψ. It is obtain by removing every linear declaration $x \hat{:} A$ from Ψ. See [4] for a formal definition.

The typing judgment for LLF^{-} has the form $\Psi \vdash M : A$ (read "M has type *A* in Ψ") and is defined in Figure 1. It relies on the auxiliary context splitting judgmen$\Psi = \Psi_1 \bowtie \Psi_2$. Due to space reasons, we shall refer the reader to [4] for a discussion of these rules.

The rewriting semantics of LLF^{-} is given by the usual P-reductions, commuting conversions (generated by the two forms of let), and, depending on one's taste, q-rules. We will only marginally be concerned with these various rules in the sequel. Definitions and properties of interest can be extrapolated fro[4]

and [6], or found in [3].

A nameless variant of LLF^- is obtained by straightforwardly extending the standard de Bruijn transformation [5]. As in the case of the X-calculus, this translation preserves typing and reductions. The resulting calculus, dLLF^-, is at the basis of our current experimentation with explicit substitutions. Space reasons prevent us from discussing it further (see [3] for more). However, its language and typing rules correspond exactly to the term fragment of the a-normal calculusxdLLF^-_σ discussed in Section 4.

3 xdLLF^-

In [6], we devised a calculus of linear explicit substitutions based on named variables. Here, we instead investigate a variant in the style of [1] that uses the de Bruijn notation (this is mainly motivated by implementation considerations). Even in this restricted setting, there are many ways to incorporate explicit substitution into LLF^- (or dLLF^-). In designing xdLLF^-, we chose to model (normal) substitutions on the structure of contexts. We mention other possibilities in Section 5.

The types of xdLLF^- are the same as LLF^-. Its term constructors are adapted from this language as done in [1]. Substitutions may contain the linear extension operator "$\hat{\ }$" to account for terms to be substituted for linear variables. Since de Bruijn numbers are positional indices in a substitution (and a context), we use "_" to mark a term that has already been linearly substituted. Terms and substitutions are defined by the following grammar:

Terms:

t ::=	1		(variable *indices)*
	0		(additive unit)
|	$\langle t_1, t_2\rangle$	$\mathsf{fst}\ t \mid \mathsf{snd}\ t$	(additive *pairs*)
|	$\bullet$	$\mathsf{let}\bullet\ t_1\ \mathsf{in}\ t_2$	(multiplicative unit)
|	$t_1 \otimes t_2$	$\mathsf{let}\otimes\ t_1\ \mathsf{in}\ t_2$	(multiplicative pairs)
|	$\hat{\lambda}_A.\, t$	$t_1 \hat{} t_2$	(linear functions)
|	$\lambda_A.\, t$	$t_1\, t_2$	(intuitionistic functions)
	$t[\sigma]$		(substitution application)

Substitutions:

σ ::=	Id	(identity)
|	$\uparrow$	(shift)
|	$t \hat{} \sigma$	(linear *extension)*
|	$\_ \hat{} \sigma$	*(used* linear *extension)*
|	$t\,.\,\sigma$	*(intuitionistic extension)*
|	$\sigma_1 \circ \sigma_2$	(composition)

In addition to t and σ, we will use s and τ to denote xdLLF^- terms and substitutions, respectively. Contexts in xdLLF^- are the nameless variant of LLF^- contexts, with again the marker "_" to account for the positional nature of de Bruijn indices when dealing with used assumptions.

$$\textit{Contexts:} \qquad \Gamma \ ::= \ \mid \ \Gamma \hat{,} A \ \mid \ \Gamma \hat{,} \_ \ \mid \ \Gamma, A$$

As in LLF^-, we write $\overline{\Gamma}$ to indicate the intuitionistic portion of Γ. It is obtain by replacing every linear assumption with "_".

The typing judgments for terms and substitutions are denoted $\Gamma \vdash_{\mathsf{xd}} t : A$ (read *"t has type A in* Γ") and $\Gamma \vdash_{\mathsf{xd}} \sigma : \Gamma'$ (read *"a maps terms from* Γ' *to* Γ"), respectively. As for LLF^-, their definition relies on the auxiliary context splitting judgment $\Gamma = \Gamma_1 \bowtie \Gamma_2$. The rules for these three judgments are displayed in Figure 2.

Rewrite rules in the β, commuting and possibly η families are adapted from LLF^-. As we said, we will not deal with them in this paper (see [3, 6] for more on this topic).

At this point, papers on explicit substitutions typically present a long list of a-reductions aimed at confining substitution application and composition to specific positions in a term and a substitution,

The Costs and Benefits of Java Bytecode Subroutines

Stephen N. Freund*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
freunds@cs.stanford.edu

September 20, 1998

Abstract

Java bytecode subroutines are used to compile the Java source language `try-finally` construct into a succinct combination of special-purpose instructions. However, the space saved by using subroutines, in comparison to simpler compilation strategies, comes at a substantial cost to the complexity of the bytecode verifier and other parts of the Java Virtual Machine. This paper examines the trade-offs between keeping subroutines and eliminating them from the Java bytecode language. We compare the cost of formally specifying the bytecode verifier and implementing the Java Virtual Machine in the presence of subroutines to the space saved by using them when compiling a set of representative Java programs.

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. We refer to this bytecode language as JVML.

Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. As has been demonstrated elsewhere, the correctness of the bytecode verifier is critical to guarantee the security of the Java Virtual Machine [DFW96]. As a step towards obtaining a correct, formal specification for the verifier, we are currently developing a specification of *statically correct bytecode* for a large fragment of JVML in the form of a type system. This system encompasses classes, interfaces, methods, constructors, exceptions, subroutines, and arrays. While still not the complete JVML, the type system for this subset contains all of the difficult static analysis problems faced by the bytecode verifier.

The most difficult and time-consuming part of our work has been handling subroutines effectively. Subroutines are mainly used to allow efficient implementation of the `try-finally` construct from the Java language. Our general approach for modeling and type checking subroutines is based on a type

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, ONR MURI Award N00014-97-1-0505, and a NSF Graduate Research Fellowship. To appear in the Workshop on Formal Underpinnings of Java, OOPSLA 98.

```
void f () {
  try {
    something0 ;
  } finally {
    done();
  }
}
```

Figure 1: A method using a try-finally statement.

system developed by Stata and Abadi [SA98a]. Even with the knowledge and techniques acquired from their earlier work, extending this type system and proofs to include sound type checking for exceptions, object initialization, and other elements of JVML was challenging.

Verifier implementations based on the current Java Virtual Machine Specification[LY96] have fared no better in handling the complexities which seem to be inherent in the analysis of subroutines. For example, several published inconsistencies and bugs, some of which lead to potential security loopholes, may be attributed to earlier versions of the Sun verifier incorrectly checking subroutines and their interactions with other parts of JVML [DFW96, FM98].

Given the important role of the verifier in the Java paradigm and the many difficulties in both specifying and implementing correct verification methods for subroutines, a natural question to ask is whether or not the benefits of subroutines justify the specification and implementation costs. Eliminating subroutines would not affect the semantics of Java. The only difference would be the compilation strategy for methods which use try-finally or other statements currently compiled using subroutines. The most straightforward way to translate a try-finally block of Java code into JVML without subroutines requires some amount of code duplication, with an exponential blow up in code size in the worst case. Clearly, removing subroutines from JVML would greatly simplify the verifier, as well as possible implementations of other parts of the Java Virtual Machine, such as type-precise garbage collectors [ADM98]. However, we know of no other study to date quantifying the benefits of subroutines and how much code size is actually saved by using subroutines in typical programs.

In this paper, we examine the impact of subroutines on the formal specification and implementation of the bytecode verifier, on the implementation of other parts of the Java Virtual Machine, and on code size for representative programs drawn from a variety of sources. Our analysis shows that the space saved by using subroutines is negligible and that the theoretically possible exponential increase in size does not occur in the programs studied. The added complexity to the verifier and the Java Virtual Machine far outweighs any benefit of subroutines.

Section 2 describes Java bytecode subroutines, how they are used in the compilation of Java programs, and the difficulties in verifying programs which use them. Section 3 presents measurements on the costs and benefits of subroutines, and Section 4 contains a discussion of these results and some concluding remarks.

2 Bytecode Subroutines

This section describes JVML subroutines and the Java language construct which they were designed to implement, the try-finally statement. We also discuss how subroutines may be used to compile

```
Method void f()
    // try block
  0 aload_0                                          // load this
  1 invokevirtual #5 <Method void something()>       // call something()
  4 jsr 14                                           // execute finally code
  7 return                                           // exit normally
    // exception handler for try block
  8 astore_1                                         // store exception
  9 jsr 14                                           // execute finally code
 12 aload_1                                          // load exception
 13 athrow                                           // rethrow exception
    // subroutine for finally block
 14 astore_2                                         // store return address
 15 aload_0                                          // load this
 16 invokevirtual #4 <Method void done()>            // call done()
 19 ret 2                                            // return from subroutine

Exception table:
   from   to target type
     0     4     8   any
```

Figure 2: The translation of the program in Figure 1 into JVML.

synchronized statements and conclude this section by describing the major difficulties in verifying bytecode subroutines.

2.1 try-finally Statements

Subroutines were designed to allow space efficient compilation of the finally clauses of exception handlers in the Java language. The details of Java exception handling facilities appear in [GJS96]. Subroutines share the same activation record as the method which uses them, and they can be called from different locations in the same method, enabling all locations where finally code must be executed to jump to a single subroutine containing that code. Without subroutines, the code from the finally block of an exception handler must be duplicated at each point where execution may escape from the handler, or some more complicated compilation technique must be used.

Figure 1 contains a sample program using a try-finally statement. There are two ways in which execution may exit from the exception handler. Either the end of the try block is reached or an exception is thrown. In both cases, the code in the finally block must be executed. Figure 2 shows the bytecode translation for this program. At the two points where execution may exit the try block, the jsr instruction is used to jump to line 14, the beginning of the subroutine containing the translation of the code from the finally block. As part of the jump, the return address is pushed onto the operand stack. This return address is stored in a local variable, and, as in line 19, the ret instruction causes a jump back to that address.

Without subroutines, the simplest way to compile this try-finally statement is to duplicate the body of the finally block at line 4 and at line 9. A translation of the program in Figure 1 that does not use subroutines is shown in Figure 3. In most cases, eliminating a subroutine and duplicating the code in this fashion results in a blow up of code size proportional in the number of calls to the subroutine. However, in the case that one subroutine calls another, the situation is

```
Method void f ()
     // try  block
   0 aload_0                                          // load this
   1 invokevirtual #5 <Method void something()>       // call something()
     // first copy of subroutine
   4 aload_0                                          // load this
   5 invokevirtual #4 <Method void done()>            // call done()
   8 return                                           // exit normally
     // exception handler for try block
   9 astore_1                                         // store exception
     // second copy of subroutine
  10 aload_0                                          // load this
  11 invokevirtual #4 <Method void done()>            // call done()
  14 aload_1                                          // load exception
  15 athrow                                           // rethrow exception

Exception table:
   from   to target type
     0     4     9   any
```

Figure 3: The translation of the program in Figure 1 into JVML without using subroutines.

much worse, and not using subroutines results in a blow up in code size exponential in the depth of the nesting of calls. Subroutines nested in this way occur when one try-finally statement is placed in the finally block of another.

There are other implementation strategies which eliminate subroutines but which may fare better in the case when they are nested. We briefly describe one of the these strategies in Appendix A, but the rest of this paper compares subroutines to only the simple code duplication strategy. As we demonstrate below, the most straightforward translation strategy seems to be suitable for all cases that appear in practice, and these more complex techniques are not required.

2.2 synchronized **Statements**

Subroutines have also proved useful in the compilation of synchronized statements. An example of a synchronized statement is shown in Figure 4. In the body of the while loop, the function must first acquire the lock on object o before executing the code guarded by the synchronized statement. The lock on o must then be released at the end of the synchronized block of code and, also, at any other point at which execution escapes from the synchronized statement. In Figure 4, this includes releasing the lock at both the continue and the break statements, as well as in the event that an exception is thrown while executing the body of the synchronized statement. A subroutine may be used to avoid duplicating the code to release the lock on o at all escape points in much the same way as they are used in the try-f inally statement.

2.3 Verifying Subroutines

The flexibility of the subroutine mechanism makes bytecode verification of subroutines difficult for two main reasons:

Part VI

Spatial Control

William Spears and *Diana Gordon:* "Using Artificial Physics to Control Agents", to appear in the Proceedings of the IEEE International Conference on Information, Intelligence, and Systems — ICIIS'99, Washington, DC, 1–3 November 1999.

Full paper: http: //www.aic.nrl.navy.mil/~gordon/papers/ap.iciis99.ps

Diana Gordon, William Spears, Oleg Sokolsky, and Insup Lee: "Distributed Spatial Control and Global Monitoring of Mobile Agents", to appear in the Proceedings of the IEEE International Conference on Information, Intelligence, and Systems — ICIIS'99, Washington, DC, l-3 November 1999.

Full paper:
http://www.aic.nrl.navy.mil/~gordon/papers/apmac.iciis99.ps

Using Artificial Physics to Control Agents

William M. Spears and Diana F. Gordon*
Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory, Code 5514
Washington, D.C. 20375
spears@aic.nrl.navy.mil

To appear **in** *the Proceedings of the IEEE International Conference* on *Information, Intelligence, and Systems (ICIIS'99)*

Abstract

We introduce a novel framework called "artificial physics", which is used to provide distributed control of large collections of agents. The agents react to artificial forces that are motivated by natural physical laws. This framework provides an effective mechanism for achieving self-assembly, fault-tolerance, and self-repair. Examples are shown for various regular geometric configurations of agents. A further example demonstrates that self-assembly via distributed control can also perform distributed computation.

Introduction

The objective of this research is the distributed control of agents that range in scale from neurons, nanobots, or micro-electromechanical systems (MEMS) to micro-air vehicles (MAVs) and satellites e.g., see (Carlson. Gupta, & Hogg 1997). Agents can be physical or virtual (e.g., softbots), mobile or immobile. Agents generally have sensors and effectors. An agent's sensors perceive the world (including other agents) and an agent's effectors make changes to that agent or the world (including other agents). Often, agents can only sense and affect nearby agents; thus the problem is usually one of "local" control. Sometimes control is also guided by global constraints and interactions.

Of course, one of the biggest problems is that we often don't know how to create the proper control rules. Not only do we want the desired global behavior to emerge from the local interaction between agents (i.e., self-assembly or self-organization), but we also would like there to be some measure of fault-tolerance i.e., the global behavior degrades very gradually if individual agents are damaged. Self-repair is also desirable, where the system repairs itself after being damaged.

The principles of self-assembly, fault-tolerance, and self-repair are precisely those exhibited by natural systems. This leads us to the hypothesis that many answers to distributed control may lie in the examination of the natural laws of physics.

A recent research thrust that is based on natural physics suggests even more strongly the close connection between physics and distributed control. This exciting research thrust is the development of alternative distributed forms of computing based on nature, such as quantum computing, molecular computing, and computing with DNA e.g., see (Adleman 1998; Gershenfeld & Chuang 1998). Such computing engines are a direct result of the natural laws of physics. In the natural world small entities (quantum bits, molecules, etc.) exert forces on other entities and respond to forces from other entities. Generally the only forces that matter are those from nearby entities, thus the computation is performed via so-called "local" interactions. However, sometimes the computations are also guided by global constraints and interactions.

Clearly the fields of natural distributed computation and distributed control are related. Both fields involve the study of large numbers of entities (or agents) undergoing changes (or performing changes) due to global constraints and local interactions from nearby entities. The main difference is in the forces that control the entities. The forces in natural distributed computing are tied directly to physical laws. The forces in distributed control are the result of man-made rules.

This paper proposes a general framework for distributed control in which "artificial physics" (AP) forces control agents. We use the term "artificial" because although we will be motivated by natural physical forces, we are not restricted to only natural physical forces. Clearly, the agents aren't really subject to real forces, but they can **act** as if the forces are real. Thus the agent's sensors must see enough to allow it to compute the forces to which it is reacting. The agent's effectors must allow it to respond to this perceived force.

We see several potential advantages to this approach. First, in the real physical world, collections of small entities yield surprisingly complex behavior from very simple interactions between the entities. Thus there is a precedent for believing that complex control can be achieved through simple local interactions. This is required for very small agents (such as neurons or nanobots), since their sensors and effectors will

*Partially supported by ONR N00014-99-WR20010 in association with the ONR Semantic Consistency MURI.

necessarily be primitive. Two, since the approach is largely independent of the size and number of agents, the results should scale well to larger agents and larger sets of agents. Finally, we believe that this approach will tighten the connection between control and computation, potentially yielding new insights into computation or yielding new computational algorithms.

Framework

The motivation for this work stems from a desire for swarms of micro-air vehicles (MAVs) to form various regular geometric configurations – thus we will focus on mobile physical agents. Our approach treats agents as physical particles, which could range in size from nanobots to satellites. A simple but realistic physical simulation of the particles' behavior was built. Particles exist in two dimensions (we see little difficulty in generalizing to three dimensions) and are considered to be point-masses. Each particle i has position p = (x, y) and velocity v = (v_x, v_y). We use a discrete-time approximation to the continuous behavior of the particles, with time-step At. At each time step, the position of each particle undergoes a perturbation Ap. The perturbation depends on the current velocity Ap = $v\Delta t$. The velocity of each particle at each time step also changes by Δv. The change in velocity is controlled by the force on the particle Au = $F\Delta t/m$, where m is the mass of that particle and F is the force on that particle. A frictional force is included, for self-stabilization.

For MAVs, the initial conditions are similar to those of a "big bang" – the MAVs are assumed to be released from a canister dropped from a plane, then they spread outwards until a desired geometric configuration is obtained. This is simulated by using a two dimensional Gaussian random variable to initialize the positions of all particles (MAVs). Velocities of all particles are initialized to be 0.0, and masses are all 1.0 (although the framework does not require this). An example initial configuration for 200 particles is shown in Figure 1.

Figure 1: The initial creation of the universe at t = 0.

Given the initial conditions and some desired global behavior, then. we must define what sensors, effectors, and force F laws are required such that the desired behavior emerges. We explore this in the next few sections, for different geometric configurations.

Creating Hexagonal Lattices

The example considered here is that of a swarm of MAVs whose mission is to form a hexagonal lattice, which creates an effective sensing grid. Essentially, such a lattice will create a virtual antenna or synthetic aperture radar to improve the resolution of radar images. A virtual antenna is expected to be an important future application of MAVs. Currently, the technology for MAV swarms (and swarms of other micro-vehicles such as micro-satellites) is in the early research stage. Nevertheless we are developing the control software now so that we will be prepared.

Since MAVs (or other small agents such as nanobots) have simple sensors and primitive CPUs, our goal was to provide the simplest possible control rules that require minimal sensors and effectors. At first blush, creating hexagons would appear to be somewhat complicated, requiring sensors that can calculate range, the number of neighbors, their angles, etc. However, it turns out that only range information is required. To understand this, recall an old high-school geometry lesson in which six circles of radius R can be drawn on the perimeter of a central circle of radius R (the fact that this can be done with only a compass and straight-edge can be proven with Galois theory). Figure 2 illustrates this construction. If the particles (shown as small circular spots) are deposited at the intersections of the circles, they form a hexagon.

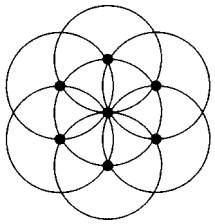


Figure 2: How circles can create hexagons.

The construction indicates that hexagons can be created via overlapping circles of radius R. To map this into a force law, imagine that each particle repels other particles that are closer than R, while attracting particles that are further than R in distance. Thus each particle can be considered to have a circular "potential well" around itself at radius R – neighboring particles will want to be at distance R from each other. The intersection of these potential wells is a form of constructive interference that creates "nodes" of very low potential energy where the particles will be likely to reside (again these are the small circular spots in the previous figure). Thus the particles serve to create the very potential energy surface they are responding to![1]

With this in mind we defined a force law $F = Gm_im_j/r^2$, where F is the magnitude of the force between two particles i and j, and r is the range between the two particles. The "gravitational constant" G is set at initialization. The force is repulsive if $r < R$ and

[1] It is important to note that the entire potential energy surface is never actually computed. Particles only compute force vectors for their current location.

attractive if $r > R$. Each particle has one sensor that can detect the range to nearby particles. The only effector is to be able to move with velocity v. To ensure that the force laws are local in nature, particles have a visual range of only $1.5R$.[2]

The initial universe of 200 particles (as shown in Figure 1) is now allowed to evolve for 1000 time steps, using this very simple force law (see Figure 3). For a radius R of 50 we have found that a gravitational constant of G = 1200 provides good results (these values for R, G, and the number of particles remain fixed throughout this paper unless stated otherwise).

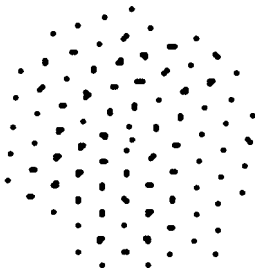


Figure 3: A good hexagonal lattice results by t = 1000.

There are a number of important observations to make about Figure 3. First, it is obvious that a reasonably well-defined hexagonal lattice has been formed from the interaction of simple local force laws that involve only the detection of distance to nearby neighbors. The hexagonal lattice is not perfect – there is a flaw near the center of the structure. Also, the perimeter is not a hexagon, although this is not surprising, given the lack of global constraints. However, many hexagons are clearly embedded in the structure and the overall structure is quite hexagonal. The second observation is that each node in the structure can have multiple particles (i.e., multiple particles can "cluster" together). Clustering was an emergent, property that we had not expected, and it provides increased robust behavior, because the disappearance (failure) of individual particles (agents) from a cluster will have minimal effect. This form of fault-tolerance is a result of the setting of G, which we explore later in this section.

The pattern of particles shown in Figure 3 is quite stable, and does not change to any significant degree as t increases past 1000. The dynamics of the evolving system (from $0 < t < 1000$) is quite fascinating (when watched on a computer screen), yet is hard to simply convey in a paper. As opposed to displaying numerous snapshots we have instead decided to focus on certain well-defined characteristics of the system that can be measured at any time step. Graphs of these characteristics yield useful insights into the system dynamics.

[2]The constant 1.5 is not chosen randomly. In a hexagon, if a nearby neighbor is further than R away, it is $\geq \sqrt{3}R$ away. We wanted the force laws to be as local as possible.

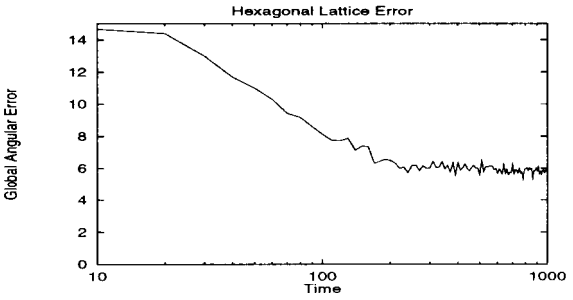


Figure 4: The average angular error in the structure as t increases. The log scale emphasizes early behavior.

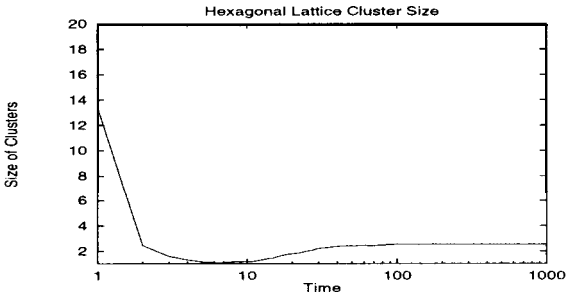


Figure 5: The size of the clusters as t increases.

The first characteristic we examined is motivated by our desire to have the global structure contain as few errors as possible, in the sense that the orientation of the hexagonal lattice should be the same everywhere throughout the lattice. To see how we can achieve a measure of this characteristic, consider choosing any pair of particles separated by $2R$. This forms a line segment. Then choose any other pair of particles also separated by $2R$, forming another line segment. Measure the angle between the two line segments. For a hexagonal lattice, this angle should be close to some multiple of 60". The **error** is the absolute value of the difference between the angle and the closest multiple of 60. The maximum error is 30" and the minimum is 0". We averaged this over all distinct pairs of particle pairs, and displayed the average error for every ten time steps. The results are shown in Figure 4.[3]

Since error ranges from 0" to 30°, we expect the average error at the beginning to be around 15". After that the error should decrease – the rate at which the decrease occurs is a reasonable measure of how quickly the system is stabilizing. Figure 4 shows that error decreases smoothly until about t = 200, resulting in a final error of roughly 6" over the whole structure. This is a typical result. Averaged over 40 independent runs (different starting conditions) the final error was 5.6".

[3]We use $2R$ instead of R in an attempt to smooth out local noise, since we care about global error. A particle is considered to be separated by $2R$ if $1.98R < r < 2.02R$.

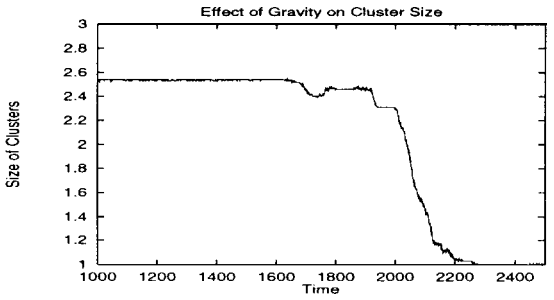


Figure 6: Cluster size drops suddenly as G is decreased linearly after t = 1000. G = $1200 - 0.5(t - 1000)$.

The second characteristic we examined is the size of clusters. For each particle i we counted the number of particles that were close to i ($0 < r < 0.2R$). We always include the particle i itself, so the minimum size of a cluster is 1.0. This was averaged over all particles and displayed for every time step. Results are shown in Figure 5. At t = 0 all particles are very close to one another, yielding a high clustering. Immediately, the particles fly apart, due to the repulsive force, so that by t = 6 the particles are all effectively separated. However. after t = 6 clusters re-emerge, with the final cluster size being around 2.5. Clearly the re-emergence of clusters serves to lower the total potential energy of the system, and the size of the re-emerged clusters depends on factors such as G, R, and the geometry of the system. A full understanding of this phenomena is beyond the scope of this paper, yet we summarize here one interesting experiment with G. We continued the previous experiment, evolving the system until t = 2500. However, after t = 1000 we lowered G by 0.5 for every time step. The results are shown in Figure 6.

We expected the average cluster size to linearly decrease with G, but in fact the behavior was much more interesting. The average cluster size remained quite constant, until about t = 2000, which is where G is 700. At this point the cluster size dramatically dropped until roughly t = 2200 (where G = 600), where the particles are separated again. This appears very similar to a phase transition in natural physics, demonstrating that AP can yield behavior very similar to that demonstrated in natural physics.

Creating Square Lattices

Given the success in creating hexagonal lattices, we were inspired to investigate other regular structures. Naturally the square lattice is an obvious choice, since (as with hexagons) squares will tile a 2D plane. The success of the hexagonal lattice hinged upon the fact that nearest neighbors are R in distance. Clearly this is not true for squares, since if the distance between particles along an edge is R, the distance along the diagonal is $\sqrt{2}R$. The problem is that the particles have no way of knowing whether their relationship to neighbors is along an edge or along a diagonal.

Figure 7: Forming a square with particles of two spins.

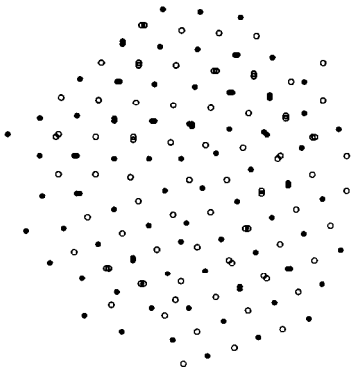


Figure 8: The 200 particles form a square lattice by ♦ = 4000. There are large global flaws in the lattice.

Once again it would appear as if we would need to know angles or the number of neighbors to solve this difficulty. In fact, a much simpler approach will do the trick. Suppose that at creation each particle is given another attribute, which we will call "spin". Half of the particles will be initialized to be spin "up", while the other half are initialized to be spin "down". Spins do not change during the evolution of the system.[4]

Now consider the square depicted in Figure 7. Particles that are spin up are open circles, while particles that are spin down are filled circles. Note that particles of unlike spin are distance R from each other, while particles of like spin are distance $\sqrt{2}R$ from each other. This "coloring" of the particles extends to square lattices, with alternating spins along the edges of squares, and same spins along the diagonals.

The construction in Figure 7 indicates that square lattices can be created if particles can sense not only range to neighbors, but also the spins of their neighbors. Thus the sensors need to be able to detect one more bit of information, spin. We use the same force law as before: $F = Gm_im_j/r^2$. In this case, however, the range $_r$ is renormalized to be $r/\sqrt{2}$ if the two particles have the same spin. Then once again the force is repulsive if $r < R$ and attractive if $r > R$. The only effector is to be able to move with velocity v. To ensure that the force laws are local in nature, particles can not even see or respond to other particles that are greater than $1.7R$ in distance.[5]

The initial universe of 200 particles is allowed to

[4] Spin is merely a particle label and has no relation to the rotational spin used in navigation templates (Slack 1990).

[5] The constant is 1.7 if particles have like spin and 1.3

Distributed Spatial Control, Global Monitoring and Steering of Mobile Physical Agents*

Diana Gordon and William Spears
Navy Center for Applied Research in AI
Naval Research Laboratory, Code 5514
Washington, D.C. 20375
gordon@aic.nrl.navy.mil

Oleg Sokolsky and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

To appear in *the Proceedings of the IEEE International* **Conference on** *Information, Intelligence, and Systems (ICIIS 99)*

Abstract

In this paper, we combine two frameworks in the context of an important application. The first framework, called "artificial physics," is described in detail in a companion paper by Spears and Gordon (1999). The purpose of artificial physics is the distributed spatial control of large collections of mobile physical agents. The agents can be composed into geometric patterns (e.g., to act as a sensing grid) by having them sense and respond to local artificial forces that are motivated by natural physics laws. The purpose of the second framework is global monitoring of the agent formations developed with artificial physics. Using only limited global information, the monitor checks that the desired geometric pattern emerges over time as expected. If there is a problem, the global monitor steers the agents to self-repair. Our combined approach of local control through artificial physics, global monitoring, and "steering" for self-repair is implemented and tested on a problem where multiple agents form a hexagonal lattice pattern.

Introduction

The objective of this research is the distributed control of large numbers of mobile physical agents to form regular geometric configurations, e.g., to act as sensing grids. During formation, the configurations are monitored by a global observer to detect whether there is a significant increase in the number of pattern violations over time. Our combined approach of distributed local control and global monitoring enables spatio-temporal coordination of the agents. The agents may range in scale from neurons, nanobots, or micro-electromechanical systems (MEMS) to micro-air vehicles (MAVs) and satellites. The example considered here is that of a swarm of MAVs whose mission is to form a hexagonal lattice, which creates an effective sensing grid. Essentially, such a lattice will create a virtual antenna or synthetic aperture radar to improve the resolution of radar images. A virtual antenna is expected to be an important future application of MAVs. Currently, the technology for MAV swarms (and swarms of other micro-vehicles such as micro-satellites) is in the early research stage. Nevertheless we are developing the control software now so that we will be prepared.

We assume agents can only sense and affect nearby agents; thus the problem is one of "local" control. The method for local control should be based on principles such as self-assembly, fault-tolerance, and self-repair. These principles are precisely those exhibited by natural systems. This leads us to look at the laws of physics for ideas on distributed control. To explore this, we have developed a general framework for distributed control in which "artificial physics" (AP) forces control agents. We use the term "artificial" because although we are motivated by natural physical forces, we are not restricted to only natural physical forces. The agents aren't really subject to real forces, but they can *act* as if the forces are real. Thus the agent's sensors will have to be able to see enough to allow it to compute the forces to which it is reacting. The agent's effectors should allow it to respond to this perceived force. For details on AP, see Spears and Gordon (1999).

We see at least two advantages to AP. First, in the real physical world, collections of small entities yield surprisingly complex behavior from very simple interactions between the entities. Thus there is a precedent for believing that complex control can be achieved through simple local interactions. This is required for very small agents (such as nanobots), since their sensors and effectors will necessarily be primitive. Two, since the approach is largely independent of the size and number of agents, the results should scale well to larger agents and larger sets of agents.

AP addresses the problem of distributed agent control via local rules. This approach, which also includes fault-tolerance and local self-repair mechanisms (Spears & Gordon 1999), may be inadequate for handling major unanticipated events. For example, if a swarm of MAVs is flying in formation, fault-tolerance and/or local self-repair capabilities could en-

*This research was supported in part by ONR N00014-97-1-0505 and ONR N00014-99-WR20010 as part of the ONR Semantic Consistency MURI, as well as NSF CCR-9619910, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466.

able recovery from minor air turbulence. On the other hand, intentional or unintentional corruption of the MAVs' control software, severe environmental conditions, or widespread mechanical failures could conceivably result in an unrecoverable problem maintaining the desired geometric formation. Therefore, we also include a global observer that monitors the progress of the formation, using the Monitoring and Checking (MaC) framework, which is described in detail in Kim *et* al. (1999). We do *not* make the strong assumption that the global observer can see the pattern – because this assumption may be infeasible for large numbers of widely distributed agents. We only assume that the observer can receive communication from the individual agents. Each agent sends an alert if it fails to satisfy its local evaluation measure. The global observer collects these alerts, and issues a general alarm if the local alerts are too frequent for too long. The general alarm might be sent to people nearby to persuade them to intervene and manually solve the problem by sending commands to the agents. Here, we assume that the general alarm suggests the need for "steering" (i.e., self-repair to recover from problems). In our approach to steering, the global observer broadcasts to the agents a global parameter change for self-repair. This restores progress toward the desired geometric configuration.

The novelties of this paper are: (1) the combination of AP with MaC, (2) the introduction of a steering method for self-repair when MaC detects a failure, and (3) experimental results that validate the usefulness of this combined approach in the context of hexagonal lattice formations. The paper begins by presenting the artificial physics framework. This is followed by a description of how AP can be used to generate hexagonal lattices. We then describe the MaC framework, and apply it to monitor the progress of forming hexagonal lattices. Finally, we present a method for steering that adjusts global parameters for self-repair. The paper concludes with some initial results, followed by related work and ideas for future research.

Artificial Physics: A Framework for Distributed Multiagent Control

Our artificial physics approach treats agents as physical particles, though their actual size may range from nanobots to satellites. A simple but realistic physical simulation of the particles' behavior was built. Particles exist in two dimensions (we see little difficulty in generalizing to three dimensions) and are considered to be point-masses. Each particle i has position $\mathrm{p} = (x, y)$ and velocity $\mathrm{v} = (v_x, v_y)$. We use a discrete-time approximation to the continuous behavior of the particles, with time-step Δt. At each time step, the position of each particle undergoes a perturbation Ap. The perturbation depends on the current velocity $\mathrm{Ap} = v\Delta t$. The velocity of each particle at each time step also changes by Δv. The change in velocity is controlled by the force on the particle $\Delta v = F\Delta t/m$, where m is the mass of that particle and F is the force on that particle. An additional simple frictional force is also always included, for self-stabilization.

Given the initial conditions and some desired global behavior, we must define what sensors, effectors, and force F laws are required such that the desired behavior emerges. We explore this for hexagonal lattices.

Creating Hexagonal Lattices

This subsection explains the construction of hexagonal lattices, e.g., for MAV sensor grids. For MAVs, the initial conditions are assumed to be similar to those of a "big bang" – the MAVs are released from a canister dropped from a plane, then they spread outwards until a desired geometric configuration is obtained. This is simulated by using a two-dimensional Gaussian random variable to initialize the positions of all particles (MAVs). Velocities of all particles are initialized to be 0.0, and masses are all 1.0 (although the framework does not require this). An example initial configuration for 150 particles is shown in Figure 1.

Figure 1: The initial creation of the universe at $t = 0$.

Since MAVs (or other small agents such as nanobots) have simple sensors and primitive CPUs, our goal is to provide the simplest possible control rules that require minimal sensors and effectors. At first blush, creating hexagons would appear to be somewhat complicated, requiring sensors that can calculate range, the number of neighbors, their angles, etc. However, it turns out that only range information is required. To understand this, recall an old high-school geometry lesson in which six circles of radius R can be drawn on the perimeter of a central circle of radius R (the fact that this can be done with only a compass and straight-edge can be proven with Galois theory). Figure 2 illustrates this construction. Notice that if the particles (shown as small circular spots) are deposited at the intersections of the circles, they form a hexagon.

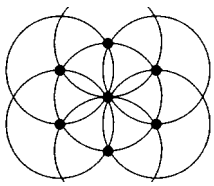


Figure 2: How circles can create hexagons.

The construction indicates that hexagons can be created via overlapping circles of radius R. To map this into a force law, imagine that each particle repels other particles that are closer than R, while attracting particles that are further than R in distance. Thus each particle can be considered to have a circular "potential well" around itself at radius R – neighboring particles will want to be at distance R from each other. The intersection of these potential wells is a form of constructive interference that creates "nodes" of very low potential energy where the particles will be likely to reside (again these are the small circular spots in the previous figure). Thus the particles serve to create the very potential energy surface they are responding to![1]

With this in mind we defined a force law $F = Gm_i m_j / r^2$, where F is the magnitude of the force between two particles i and j, and r is the range between the two particles. The "gravitational constant" G is set at initialization. The force is repulsive if $r < R$ and attractive if $r > R$. Each particle has one sensor that can detect the range to nearby particles. The only effector is to be able to move with velocity v. To ensure that the force laws are local in nature, particles can not even see or respond to other particles that are greater than $1.5R$ in distance. [2]

The initial universe of 150 particles (as shown in Figure 1) is now allowed to evolve, using this very simple force law. For a radius R of 50 we have found that a gravitational constant of G = 1200 provides good results (these values for R, G, and the number of particles remain fixed throughout this paper). Figure 3 shows the system after 35 time steps.

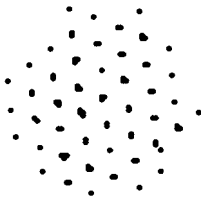


Figure 3: The 150 particles form a good hexagonal lattice by t = 35.

There are a couple of important observations to make about Figure 3. First, a reasonably well-defined hexagonal lattice has been formed from the interaction of simple local force laws that involve only the detection of distance to nearby neighbors. Also, the perimeter is not a perfect hexagon, although this is not surprising, given the lack of global constraints. However, many hexagons are clearly embedded in the structure and the overall structure is quite hexagonal. The second observation is that each node in the structure can have multiple particles (i.e., multiple particles can "cluster" together). Clustering is an emergent property that provides increased robust (fault tolerant) behavior, because the disappearance of individual agents from a cluster will have minimal effect.

Discussion

The artificial physics framework offers a number of advantages. For one, it enables large numbers of agents to self-assemble into geometric lattices. Here, we have shown the method for assembling hexagonal lattices. With a minor extension (the introduction of a "spin" attribute), agents can also self-assemble into square lattices, "open" hexagonal lattices (i.e., without an agent in the center of the hexagon), and an approximation to lattices of pentagons.[3] Furthermore, as mentioned above, fault-tolerance is a result of the emergent redundancy at nodes of the lattice. In Spears and Gordon (1999), it is shown that there is an effective offline evaluation measure of lattice quality that averages the angular error throughout the lattice. This is useful during program development. Furthermore, Spears and Gordon (1999) present effective local self-repair methods that can fill gaps in the lattice (empty nodes) and reduce the angular error.

Although AP has the desirable attributes of enabling self-assembly, fault-tolerance, and local self-repair, it cannot address all problems that the agents might encounter. In particular, although the offline measure of lattice quality provides assistance during program development, it relies on measuring angles and making geometric comparisons between agents that are far apart in the lattice. As stated earlier, we do not want agents to have to measure angles, and we cannot assume sensors that detect other agents beyond the visibility range. Therefore we require a simpler online measure of lattice quality. Furthermore, although the local self-repair methods are effective for repairing empty nodes and global flaws in angles (such as those detected by the angular error measure), they are not capable of restoring the lattice after severe disturbances that distort the shape of the perimeter. An example of a potential hazard for an MAV is air turbulence. MAVs are expected to be small (less than six inches in length, width, and height), slow (traveling 22-45 miles per hour), and light (50-70 grams). This translates into a low Reynolds number, which implies that for practical purposes inertia can be ignored and the MAVs will be especially vulnerable to air turbulence. (McMichael & Francis 1997). Our solution is to add Monitoring and Checking.

[1] It is important to note that the entire potential energy surface is never actually computed. Particles only compute force vectors for their current location.

[2] The constant 1.5 is not chosen randomly. In a hexagon, if a nearby neighbor is further than R away, it is $\geq \sqrt{3}R$ away. We wanted the force laws to be as local as possible.

[3] It is an approximation because it's impossible to generate a tiling with regular pentagons.

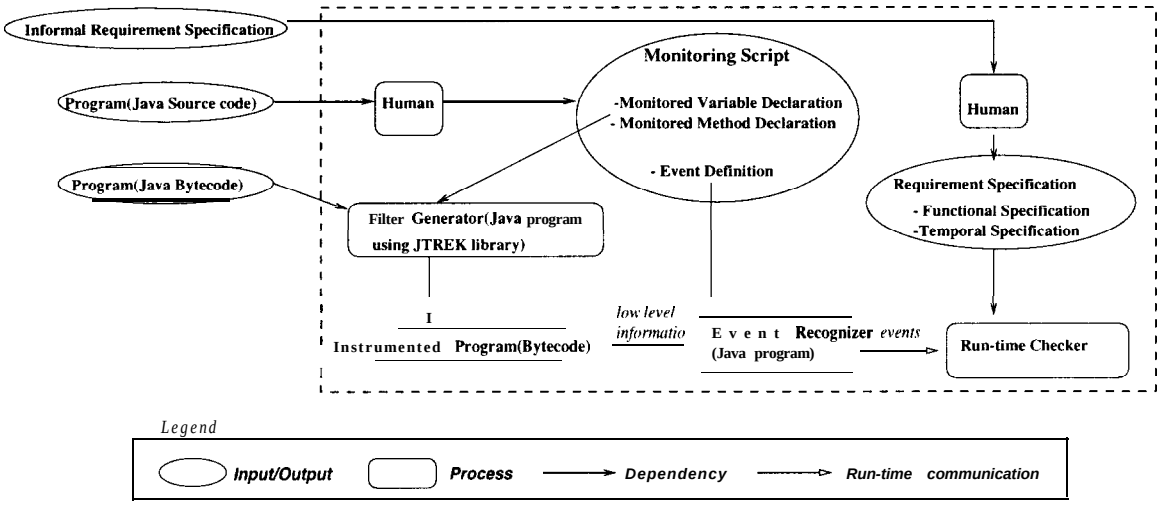


Figure 4: Overview of the MaC framework.

A Framework for Global Monitoring

The Monitoring and Checking (MaC) framework (see Figure 4) aims at run-time assurance monitoring of real-time systems. The current implementation is in Java, though the framework is generic and can apply to any language. The framework includes two main phases: (1) before the system is run, its implementation and requirement specification are used to generate run-time monitoring components; (2) during system execution, information about the running system is collected and matched against the (user-generated) requirements.

During the first phase, MaC provides a mapping between high-level events used in the requirement specification and low-level state information extracted during execution. They are related explicitly by means of a monitoring *script,* which describes how events at the requirements level are defined in terms of monitored states of an implementation. For example, in the requirements we may want to express the event that the agents are `tooClose`. The implementation, on the other hand, stores the information about proximity in a variable distance. In an execution state, this variable has a particular value. The monitoring script in this case can define the event `tooClose` as `(distance > 0.25*R) && (distance < 0.75*R)`. This definition of `tooClose` captures the notion that if neighboring particles are $\leq 0.25R$ apart then we permit this because they are in the same cluster (node); however, if they are not in the same cluster then we want them to be approximately R apart,.

The monitoring script is used to automatically generate a *filter* and an *event recognizer* for run-time monitoring. The filter is a set of program fragments that are inserted into the implementation to instrument the system. Instrumentation is performed statically directly on the code (bytecode in the case of Java). Instrumentation is automatic, which is made possible by the low-level description in the monitoring script. The essential functionality of the filter is to keep track of changes to monitored objects and send pertinent state information to the event recognizer.

The monitoring script is also used to automatically generate the event recognizer. The event recognizer detects, according to the monitoring script, occurrences of high-level events from the data received from the filter. The purpose of the event recognizer is to deliver events to a *run-time* **checker,** described below.

Also, during the first phase the user formalizes the system requirements in *a requirements specification.* The requirements in this specification are defined in terms of events (which are defined in the monitoring script). A run-time checker is produced automatically from the requirements specification. The purpose of the run-time checker is to determine at run-time whether the system is satisfying its requirements.

In summary, during the first phase the user defines a requirements specification and a monitoring script. The requirements specification defines what the user expects of the system. The monitoring script provides event definitions necessary for the requirements specification. From the monitoring script, a filter and event recognizer are automatically generated, and from the requirements specification, a run-time checker is automatically generated.

During the second (run-time) phase, the instrumented implementation is executed while being monitored. The filter sends relevant state information to the event recognizer, which detects events. These events are then relayed to the run-time checker, which checks adherence to the user-desired requirements.