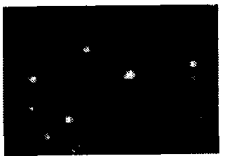

Pleiades Project

Collected Work
1997-1998

Multidisciplinary Research Initiative

Semantic Consistency in Information Exchange

Organization 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 April 1997 to August 1998. Its intent is to give the reader an overview of our accomplishments during this initial phase of the project. Therefore, rather than including complete publications, we chose to reproduce only the first four pages of each paper. In order to satisfy the legitimate curiosity of readers interested in specific articles, each paper can be integrally retrieved from the World-Wide Web through the provided URL. 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, Probabilistic Program Correctness, Programming Languages, Temporal Reasoning, and Adaptive Agents.

The *Pleiades Project*, more formally known as *Multidisciplinary Research Initiative (MURI)* on **Semantic** *Consistency* **in** *Information Exchange*, is funded by grant number N00014-97-0505 of the Organization 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 that may be implemented in different ways or designed according to different objectives or assumptions.

The current participants of the *Pleiades Project* include Iliano Cervesato (Stanford University), *Cynthia Dwork* (IBM Almaden Research Center), *Funda Ergün* (University of Pennsylvania), *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), and *Ulrich Stern* (Stanford University), and several graduate students from Cornell University, the University of Pennsylvania, and Stanford University.

Stanford, September 21st 1998
Iliano Cervesato,
John C. Mitchell.

Contents

Part I

Security Protocol Analysis

Patrick Lincoln, John Mitchell, Mark Mitchell, and *Andre Scedrov:* "A probabilistic poly-time framework for protocol analysis", in the proceedings of the fifth ACM Conference on Computer and Communications Security, San Francisco, CA, November, 1998.

Full paper: `file://www.cis.upenn.edu/pub/papers/scedrov/acmccs.ps.gz`

John C. Mitchell, Mark Mitchell, and Ulrich Stern: "Automated Analysis of Cryptographic Protocols Using Murphi", in the Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 141-153, Oakland, CA, May 1997.

Full paper: `ftp://theory.stanford.edu/pub/jcm/papers/murphi-protocols.ps`

John C. Mitchell, Vitaly Shmatikov, and Ulrich **Stern:** "Finite-State Analysis of SSL 3.0", in the Proceedings of the 7th USENIX Security Symposium, pages 201-216, San Antonio, 1998. Preliminary version presented at DIMACS Workshop on Design and Formal Verification of Security Protocols, September 1997, and distributed on workshop CD.

Full paper: `ftp://theory.stanford.edu/pub/jcm/papers/ssl-usenix.ps`

Vitaly Shmatikov and Ulrich Stern: "Efficient Finite-State Analysis for Large Security Protocols", in the Proceedings of the 11th IEEE Computer Security Foundations Workshop, pages 106-115, Rockport, MA, June 1998.

Full paper: http: //sprout. Stanf ord. `EDU/uli/secur/prio.ps . Z`

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 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

†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.

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 A4 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 ~-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 ~-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 n-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
$P\|Q$	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 A4 = N then do P (guarded command)
let x = A4 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 (n, $M\rangle$ onto the network. The expression $n(x).\ P$ matches any pair (n, m) and continues process P with x bound to value m. When $n(x).\ P$ matches a pair (n, $M\rangle$, the pair (n, $M\rangle$ is removed from the network and is no longer available to be read by another process. Evaluation of $n(x).\ P$ does not proceed unless or until a pair $\langle n, m\rangle$ 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 $n_{pasv}(x).\ \boldsymbol{P}$ as an abbreviation for n(z). $(\overline{n}\langle x\rangle \mid \boldsymbol{P})$. 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 $\overline{x}\langle a\rangle \mid n_{pasv}(x).\ \boldsymbol{P} \mid \boldsymbol{Q}$ 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

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

We illustrate the calculus by restating a simple protocol written in "the notation commonly found in the literature" where $\boldsymbol{A} \rightarrow B$ 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.***

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

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 r-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.

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

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

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

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 $AB(x).\ \overline{AB}\langle \textit{encrypt}(p_1, \textbf{\textit{kc}})\rangle$ 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

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

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 r-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 $\overline{PUBLIC}\langle p\rangle \mid \textbf{\textit{PUBLIC(g)}}$
| let a be a random number in $[1, p-1]$
    in $\overline{AB_1}\langle g^a \ \text{mod}\ \textbf{\textit{p)}}$
      $\mid BA(x).\ \overline{AB_2}\langle \mathsf{msg} * x^a \ \text{mod}\ \textbf{\textit{p)}}$
| let b be a random number in $[1, p-1]$
    in $AB_1(y).\ \overline{BA}\langle g^b \ \text{mod}\ \textbf{\textit{p)}}$

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

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

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 $\mathcal{C}[\,]$ 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

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

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

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

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 (n, m) , 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 $\nu n.$. (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 $\mathcal{C}[\,]$ and observable action v , it seems reasonable to compare the probability that $\mathcal{C}[P] \overset{eval}{\rightsquigarrow}$ v to the probability that $\mathcal{C}[Q] \overset{eval}{\rightsquigarrow}$ v. However, since a probability distribution is an infinite entity, it is not clear how to "observe" a distribution. We might run $\mathcal{C}[P]$ some number of times. count how many times v occurs, and then repeat rhe series of experiments for

Automated Analysis of Cryptographic Protocols Using Murφ

John C. Mitchell Mark Mitchell Ulrich Stern
Dept Computer Science
Stanford University
Stanford, CA 94305

Abstract

A methodology is presented for using *a general-purpose state* enumeration *tool, Murφ, to* analyze cryptographic *and security-related protocols. We illustrate the feasibility of the approach by* analyzing *the Needham-Schroeder protocol, finding a known bug in a few* seconds *of computation time, and* analyzing *variants of Kerberos* and *the faulty TMN protocol used in another comparative study. The efficiency of Murφ allows us to examine multiple runs of relatively* short *protocols. giving us the ability to detect replay attacks, or* errors resulting *from* confusion *between independent execution of a protocol by independent parties.*

1 Introduction

Encouraged by the success of others in analyzing the Needham-Schroeder public-key authentication protocol using the FDR model checker for CSP [10, 11, 13, 14], we have carried out a feasibility study for a related, but somewhat different general tool called Murφ [1], pronounced "Mur-phi". In this paper, we outline our general methodology and summarize our investigation of three protocols. First; we repeat Lowe's analysis of the Needham-Schroeder protocol, finding a violation of the correctness condition in a simplified protocol, and then failing to find a violation in a repaired version of the protocol. Next, we analyze the TMN protocol [18], first finding a simple error also identified by two of the three tools described in a comparative study by Kemmerer, Meadows and Millen [7]. (These three tools appear to require more expert guidance than our brute force state exploration tool.) After modifying our system description to eliminate the first error, our system finds a second automatically. With some minor refinement of the cryptographic model, based on general principles we present in this paper, a third run also uncovers a related RSA-specific error that is explained in [18] and also discovered by the third tool in Kemmerer, Meadows and Millen's comparative study (but not the other two tools). We also investigate Kerberos, version 5, finding a failure in a simplified version based on documentation [9], and then "verifying" a repaired version that is closer to the full implementation given in RFC-1510 [8]. One interesting aspect of the Kerberos error is that it only occurs in a system configuration that includes more than the minimal number of participants.

In general, we believe that a general-purpose tool for analyzing finite-state systems may be useful for analyzing cryptographic or security-related protocols. The main challenges that arise are:

- State-space explosion, as with other tools,
- Subtleties involving formalization of the adversary or adversaries, and
- Subtleties involving properties of the encryption primitives, which may be modeled as completely secure black-box primitives, or primitives with other algebraic or "malleability" [3] properties.

One aspect of our approach that we believe will prove useful is that it is feasible to modify a Murφ system description to reflect a situation where one or more pieces of secret information have been compromised. For example, it is easy to modify our Kerberos description to give the adversary knowledge that two clients are using the same private key, without revealing the key to the adversary. The method is illustrated in our analysis of TMN to allow the adversary to generate an encryption of *nm* from an encryption of n, for

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891, **and the National Science Foundation through grants** CCR-9303099 **and** CCR-9629754. **The views and** conclusions **contained in this document are those of the author(s) and should not be interpreted** as **representing the official policies, either expressed or implied: of the Defense Advanced Research Projects Agency, NASA, NSF or the US Government.**

any numbers n and *m*, without allowing the adversary to decrypt any messages. The fact that an adversary can compute the RSA-encryption of one message from the RSA-encryption of another, without decrypting, is an example of "malleability" [3]. Since previous analyses tend to assume non-malleability, we expect that further insight into specific protocols may be gained by taking algebraic properties of specific cryptosystems into consideration.

Some promising future directions involve automatic translation of a higher-level protocol specification language such as CAPSL into Murcp, and combined analyses using both exhaustive finitestate analysis and formal logical methods. In particular, we hope to develop better techniques for using the results of state enumeration to simplify formal correctness proofs for potentially unbounded (or non-finite) systems, and to use formal proofs of invariants to narrow the search space for state enumeration. A larger limitation, to which we have not yet turned our attention, is that we have no way of incorporating probabilistic analysis. For example, we cannot outfit our adversary with an unbiased coin and compute the probability that a randomized attack will compromise a protocol.

Contents:

2 Outline of the methodology

Our general methodology is similar to the approach used in CSP model checking [10, 14] of cryptographic protocols. However, there are some differences between Murcp and FDR.

2.1 The Murcp Verification System

Murcp [1] is a protocol verification tool that has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [2, 16, 19].

To use Murφ for verification, one has to model the protocol in the Murcp language and augment this model with a specification of the desired properties. The Murφ system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murcp language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The state of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by rules. Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e. the rule is enabled) and may change global variables. Most Murcp models are nondeterministic; usually more than one rule is enabled in each state. For example, in a model of a cryptographic protocol, the intruder typically has the nondeterministic choice of several messages to replay.

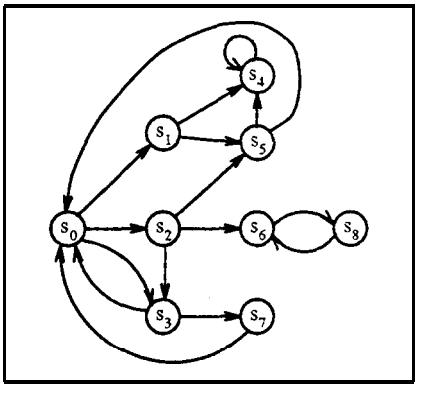

Figure 1. Sample state graph

Figure 1 shows a simple sample state graph with nine states ($s_0, \ldots, s_8$). The outgoing arcs in each state correspond to the rules that are enabled in that state. While a simulator chooses an outgoing arc at random, a verifier explores all reachable states from a given startstate (s_0).

Murcp has no explicit notion of processes. Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel composition* of two processes in Murcp is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of

asynchronous, interleaving concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murcp language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a down-scaled (finite state) version of the protocol. Murcp can only guarantee correctness of the down-scaled version of the protocol, but not correctness of the general protocol. For example, in the model of the Needham Schroeder protocol, the numbers of initiators and responders are scalable and defined by constants.

The Murcp verifier supports automatic *symmetry* reduction of models by special language constructs [4]. For example, in the Needham Schroeder protocol, if we have two initiators A_1 and A_2, the state where initiator A_1 has started the protocol and A_2 is idle is – for verification purposes – the same as the state where A_1 is idle and A_2 has started the protocol.

The desired properties of a protocol can be specified in Murcp by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murcp prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

There are two main differences between Murcp and FDR. First, while communication is supported in FDR by the CSP notions of channels and events, it is modeled by shared variables in Murcp. Second, Murφ currently implements a richer set of methods for increasing the size of the protocols that can be verified, including symmetry reduction [4], hash compaction [17], reversible rules [5], and repetition constructors [6]. In addition, there is a parallel version of the Murcp verifier [15]. Although available for internal use, the latter three techniques are not yet in the public Murcp release.

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. *Formulate the protocol.* This generally involves simplifying the protocol by identifying the key steps and primitives. However, the Murcp formulation of a protocol is more detailed than the high-level descriptions often seen in the literature. The most significant issue is to decide exactly which messages will be accepted by each participant in the protocol (see Section 6 for further discussion). Since Murcp communication is based on shared variables, it is also necessary to define an explicit message format, as a Murcp type.

2. *Add an adversary to the system.* We generally assume that the adversary is a participant in the system, capable of initiating communication with an honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:

   - overhear every message, remember all parts of each message, and decrypt cyphertext when it has the key,
   - intercept (delete) messages,
   - generate messages using any combination of initial knowledge about the system and parts of overheard messages.

   Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of effecting other participants. This is discussed in more detail in Section 6.

3. State *the desired correctness condition.* We have generally found it easy to state correctness conditions, but we have no reason to believe that there are not other protocols where this step could prove subtle.

4. *Run the protocol* for some specific choice of system size parameters. Speaking very loosely, systems with 4 or 5 participants (including the adversary) and 3 to 5 intended steps in the original protocol (without adversary) are easily analyzed in minutes of computation time using a modest workstation. Doubling or tripling these numbers; however, may cause the system to run for many hours, or terminate inconclusively by exceeding available memory.

5. Experiment *with alternate formulations and repeat.* In our examples, which were known to be incorrect (except possibly Kerberos), we have repaired the detected error, either by strengthening the protocol or, where this did not seem feasible, redirecting the efforts of the adversary. In cases where a protocol appears correct, it also may be interesting to investigate the consequences of strengthening the adversary, possibly by providing some of the "secret" information.

Clearly there are many calls for creativity and good judgment; this is not in any way an automatic procedure that could be carried out routinely from a high-level description of a protocol. However, as we gain more experience with the method, we anticipate development of certain tools that will make the process easier to carry out.

3 Needham-Schroeder Exchange with Public Keys

3.1 Overview of the Protocol

The Needham-Schroeder Public-Key Protocol [12] aims at mutual authentication between an initiator ***A*** and a responder ***B,*** i.e. both initiator and responder want to be assured of the identity of the other.

As in [10], we only study a simplified version of the protocol, which can be described by the following three steps.

$$\begin{aligned} A \text{ - } B &: \{N_a,\ A\}_{K_b} \\ B \text{ - } A &: \{N_a,\ N_b\}_{K_a} \\ A \text{ - } B &: \{N_b\}_{K_b} \end{aligned}$$

The initiator ***A*** sends a nonce N_a (i.e. a newly generated random number) and its identifier to reponder ***B,*** both encrypted with ***B's*** public key K_b. Responder ***B*** decrypts the message and obtains knowledge of N_a. It then generates a nonce N_b itself and sends both nonces encrypted with ***A's*** public key to ***A.*** Initiator ***A*** decrypts the message and concludes that it is indeed talking to ***B,*** since only ***B*** was able to decrypt ***A's*** initial message containing nonce N_a; ***B*** is authenticated. In a corresponding fashion, ***A*** is authenticated after the third step of the protocol. (This is not entirely correct, though, as we shall see.)

3.2 Modeling the Protocol

Due to space constraints, we look only at the initiator part of the model in detail. The data structures for the initiator are as follows:

```
const
  NumInitiators:  1;
type
  InitiatorId: scalarset (NumInitiators) ;
  InitiatorStates: enum{I_SLEEP,I_WAIT,I_COMMIT};
  Initiator:  record
    state: InitiatorStates;
    responder: AgentId;
  end;
var
  ini: array [InitiatorId] of  Initiator;
```

The number of initiators is scalable and defined by the constant NumInitiators. The type InitiatorId can be thought of as a subrange 1.. .NumInitiators with the difference that automatic symmetry reduction is invoked by declaring this type a scalarset. The state of each initiator is stored in the array ini. In the startstate statement of the model, the local state (stored in field state) of each initiator is set to I_SLEEP, indicating that no initiator has started the protocol yet.

The behavior of an initiator is modeled with two Murφ rules. In the first rule, the initiator starts the protocol by sending the initial message to some agent and changes its local state from I_SLEEP to I-WAIT. The second rule models the reception and checking of the reply from the agent, the commitment and the sending of the final message.

The Murφ code of the first rule looks as follows:

```
ruleset i: InitiatorId do
  ruleset j: AgentId do
    rule "initiator starts protocol"
      ini[i] . state = I_SLEEP &      -- condition
      !ismember(j,InitiatorId) &
      multisetcount(l:net, true) < NetworkSize
    ==>
    var
      outM: Message;    -- outgoing message
    begin                           -- action
      undefine outM;
      outM.source := i;
      outM.dest  :=  j;
      ... set remaining fields of outM
      multisetadd(outM,net);
      ini[i].state      := I-WAIT;
      ini[i].responder := j;
    end;
  end;
end;
```

In this code segment, the rule in enclosed by two ruleset statements. These statements make the rule scalable: it is instantiated for each initiator of type InitiatorId and for each agent of type AgentId. Thus, when one changes the constant NumInitiators, the number of rules automatically adapts to this change. The identifiers of the initiator and the agent of a particular instantiation are assigned to the local variables i and j, respectively, and can be used in the rule.

The condition of the rule is that initiator i is in the local state I_SLEEP, that agent j is not an initiator (and hence either reponder or intruder). and that there is space in the network for an additional message. The network is modeled by the shared variable net. Each network cell can hold one message of the protocol. In

Finite-State Analysis of SSL 3.0

John C. Mitchell Vitaly Shmatikov Ulrich Stern

Computer Science Department
Stanford University
Stanford, CA 94305

{jcm, shmat, uli}@cs.stanford.edu

Abstract

The Secure Sockets Layer (SSL) protocol is analyzed using a ***finite-state enumeration tool*** called *Murφ*. ***The*** analysis ***is presented*** using ***a sequence*** *of* incremental ***approximations to the SSL 3.0 handshake protocol. Each simplified protocol is "model-checked" using Murφ, with the next protocol in the sequence obtained by*** *correcting* ***errors that Murφ finds automatically. This process identifies the main shortcomings in SSL 2.0 that led to the design*** *of* ***SSL 3.0, as well as a*** *few* ***anomalies in the protocol that is used to resume a session in SSL 3.0. In addition to*** some ***insight into SSL, this study demonstrates the feasibility*** *of* ***using formal methods to analyze com-*** mercial ***protocols.***

1 Introduction

In previous work [9], a general-purpose finite-state analysis tool has been successfully applied to the verification of small security protocols such as the Needham-Schroeder public key protocol, Kerberos, and the TMN cellular telephone protocol. The tool, Murφ [3, 10], was designed for hardware verification and related analysis. In an effort to understand the difficulties involved in analyzing larger and more complex protocols, we use Murφ to analyze the SSL 3.0 handshake protocol. This protocol is important, since it is the de facto standard for secure Internet communication, and a challenge, since it has more steps and greater complexity than the other security protocols analyzed using automated finite-state exploration. In addition to demonstrating that finite-state analysis is feasible for protocols of this complexity, our study also points to several anomalies in SSL 3.0. However, we have not demonstrated the possibility of compromising sensitive data in any implementation of the protocol.

In the process of analyzing SSL 3.0, we have developed a "rational reconstruction" of the protocol. More specifically, after initially attempting to familiarize ourselves with the handshake protocol, we found that we could not easily identify the purpose of each part of certain messages. Therefore, we set out to use our analysis tool to identify, for each message field, an attack that could arise if that field were omitted from the protocol. Arranging the simplified protocols in the order of increasing complexity, we obtain an incremental presentation of SSL. Beginning with a simple, intuitive, and insecure exchange of the required data, we progressively introduce signatures, hashed data, and additional messages, culminating in a close approximation of the actual SSL 3.0 handshake protocol.

In addition to allowing us to understand the protocol more fully in a relatively short period of time, this incremental reconstruction also provides some evidence for the "completeness" of our analysis. Specifically, Murφ exhaustively tests all possible interleavings of protocol and intruder actions, making sure that a set of correctness conditions is satisfied in all cases. It is easy for such analysis to be "incomplete" by not checking all of the correctness conditions intended by the protocol designers or users. In developing our incremental reconstruction of SSL 3.0, we were forced to confirm the importance

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891, Office of Naval Research grant N00014-97-1-0505, Multidisciplinary University Research Initiative "Semantic Consistency in Information Exchange", National Science Foundation grant CCR-9629754, and the Hertz Foundation. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, ONR, NSF or the US Government.

of each part of each message. In addition, since no formal or high-level description of SSL 3.0 was available, we believe that the description of SSL 3.0 that we extracted from the Internet Draft [6] may be of interest.

Our analysis covers both the standard handshake protocol used to initiate a secure session and the shorter protocol used to resume a session [6, Section 5.51. Murcp analysis uncovered a weak form of version rollback attack (see Section 4.9.2) that can cause a version 3.0 client and a version 3.0 server to commit to SSL 2.0 when the protocol is resumed. Another attack on the resumption protocol (described in Sections 4.8 and 4.9.1) is possible in SSL implementations that strictly follow the Internet draft [6] and allow the participants to send application data without waiting for an acknowledgment of their Finished messages. Finally, an attack on cryptographic preferences (see Section 4.6) succeeds if the participants support weak encryption algorithms which can be broken in real time. Apart from these three anomalies, we were not able to uncover any errors in our final protocol. Since SSL 3.0 was designed to be backward-compatible, we also implemented and checked a full model for SSL 2.0 as part of the SSL 3.0 project. In the process, Murcp uncovered the major problems with SSL 2.0 that motivated the design of SSL 3.0.

Our Murcp analysis of SSL is based on the assumption that cryptographic functions cannot be broken. For this and other reasons (discussed below), we cannot claim that we found all attacks on SSL. But our analysis has been efficient in helping discover an important class of attacks.

The two prior analyses of SSL 3.0 that we are aware of are an informal assessment carried out by Wagner and Schneier [14] and a formal analysis by Dietrich using a form of belief logic [2]. (We read the Wagner and Schneier study before carrying out our analysis, but did not become aware of the Dietrich study until after we had completed the bulk of our work.) Wagner and Schneier comment on the possibility of anomalies associated with resumption, which led us to concentrate our later efforts on this area. It is not clear to us at the time of this writing whether we found any resumption anomalies that were not known to these investigators. However, in email comments resulting from circulation of an earlier document [13], we learned that while our second anomaly was not noticed by Wagner and Schneier, it was later reported to them by Michael Wiener. Neither anomaly seems to have turned up in the logic-based study of [2].

A preliminary report on this study was presented in a panel at the September 1997 DIMXCS Workshop on Design and Formal Verification of Security Protocols and appears on the web site (`http://dimacs.rutgers.edu/Workshops/Security/`) and CD ROM associated with this workshop. Our study of resumption was carried out after our workshop submission and is not described in the workshop document.

2 Outline of the methodology

Our general methodology for modeling security protocols in Murcp is described in our previous paper [9], and will be only outlined in this section. The basic approach is similar to the CSP approach to model checking of cryptographic protocols described in [8, 11].

2.1 The Murcp verification system

Murφ [3] is a protocol or, more generally, finite-state machine verification tool. It has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [4, 12, 15]. The purpose of finite-state analysis, commonly called "model checking," is to exhaustively search all execution sequences. While this process often reveals errors, failure to find errors does not imply that the protocol is completely correct, because the Murcp model may simplify certain details and is inherently limited to configurations involving a small number of, say, clients and servers.

To use Murcp for verification, one has to model the protocol in the Murcp language and augment this model with a specification of the desired properties. The Murcp system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murcp language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The ***state*** of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by *rules*.

Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e., the rule is enabled) and typically changes global variables, yielding a new state. Most Murφ models are nondeterministic since states typically allow execution of more than one rule. For example, in the model of the SSL protocol, the intruder (which is part of the model) usually has the nondeterministic choice of several messages to replay.

Murcp has no explicit notion of ***processes.*** Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel* ***composition*** of two processes in Murcp is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of ***asynchronous,*** interleaving concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murcp language supports scalable models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a down-scaled (finite state) version of the protocol. Murcp can only guarantee correctness of the down-scaled version of the protocol, but not correctness of the general protocol. For example, in the model of the SSL protocol, the numbers of clients and servers are scalable and defined by constants.

The desired properties of a protocol can be specified in Murcp by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murcp prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. ***Formulate the protocol.*** This generally involves simplifying the protocol by identifying the key steps and primitives. The Murcp formulation of a protocol, however, is more detailed than the high-level descriptions often seen in the literature, since one has to decide exactly which messages will be accepted by each participant in the protocol. Since Murcp communication is based on shared variables, it is also necessary to define an explicit message format, as a Murcp type.
2. ***Add an*** adversary ***to the system.*** We generally assume that the adversary (or intruder) can masquerade as an honest participant in the system, capable of initiating communication with a truly honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:
   - overhear every message, remember all parts of each message, and decrypt ciphertext when it has the key,
   - intercept (delete) messages,
   - generate messages using any combination of initial knowledge about the system and parts of overheard messages.

   Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of affecting other participants.
3. ***State the desired correctness condition.*** A typical correctness criterion includes, e.g., that no secret information can be learned by the intruder. More details about the correctness criterion used for our SSL model are given in Section 3.
4. Run ***the protocol*** for some specific choice of system size parameters. Speaking very loosely, systems with 4 or 5 participants (including the adversary) and 5 to 7 intended steps in the original protocol (without adversary) are easily analyzed in minutes of computation time using a modest workstation. Doubling or tripling these numbers, however, may cause the system to run for many hours, or terminate inconclusively by exceeding available memory.
5. ***Experiment with alternate formulations and repeat.*** This is discussed in detail in Section 4.

2.3 The intruder model

The intruder model described above is limited in its capabilities and does not have all the power that a real-life intruder may have. In the following, we discuss examples of these limitations.

***No* cryptanalysis.** Our intruder ignores both computational and number-theoretic properties of cryptographic functions. As a result, it cannot perform any cryptanalysis whatsoever. If it has the proper key, it can read an encrypted message (or forge a signature). Otherwise, the only action it can perform is to store the message for a later replay. We do not model any cryptographic attacks such as brute-force key search (with a related notion of computational time required to attack the encryption) or attacks relying on the mathematical properties of cryptographic functions.

No probabilities. Murcp has no notion of probability. Therefore, we do not model "propagation" of attack probabilities through our finite-state system (e.g, how the probabilities of breaking the encryption, forging the signature, etc. accumulate as the protocol progresses). We also ignore, e.g., that the intruder may learn some probabilistic information about the participants' keys by observing multiple runs of the protocol.

No partial information. Keys, nonces, etc. are treated as atomic entities in our model. Our intruder cannot break such data into separate bits. It also cannot perform an attack that results in the partial recovery of a secret (e.g., half of the secret bits).

In spite of the above limitations, we believe that Murcp is a useful tool for analyzing security protocols. It considers the protocol at a high level and helps discover a certain class of bugs that do not involve attacks on cryptographic functions employed in the protocol. For example, Murcp is useful for discovering "authentication" bugs, where the assumptions about key ownership, source of messages, etc. are implicit in the protocol but never verified as part of the message exchange. Also, Murcp models can successfully discover attacks on plaintext information (such as version rollback attacks in SSL) and implicit assumptions about message sequence in the protocol (such as unacknowledged receipt of ***Finished*** messages in SSL). Other examples of errors discovered by finite-state analysis appear in [8, 9, 11] and in references cited in these papers.

3 The SSL 3.0 handshake protocol

The primary goal of the SSL 3.0 handshake protocol is to establish secret keys that "provide privacy and reliability between two communicating applications" [6]. Henceforth, we call the communicating applications the client (C) and the server (S). The basic approach taken by SSL is to have C generate a fresh random number (the ***secret*** or ***shared secret)*** and deliver it to S in a secure manner. The secret is then used to compute a so-called ***master secret*** (or ***negotiated cipher),*** from which, in turn, the keys that protect and authenticate subsequent communication between C and S are computed. While the SSL ***handshake protocol governs*** the secret key computation, the SSL record layer ***protocol*** governs the subsequent secure communication between C and S.

As part of the handshake protocol, C and S exchange their respective ***cryptographic*** preferences, which are used to select a mutually acceptable set of algorithms for encrypting and signing handshake messages. In our analysis, we assume for simplicity that RSA is used for both encryption and signatures, and cryptographic preferences only indicate the desired lengths of keys. In addition, SSL 3.0 is designed to be backward-compatible so that a 3.0 server can communicate with a 2.0 client and vice versa. Therefore, the parties also exchange their respective version numbers.

The basic handshake protocol consists of three messages. With the *ClientHello* message, the client starts the protocol and transmits its version number and cryptographic preferences to the server. The server replies with the *ServerHello* message, also transmitting its version number and cryptographic preferences. Upon receipt of this message, the client generates the shared secret and sends it securely to the server in the ***secret exchange message.***

Since we were not aware of any formal definition of SSL 3.0, we based our model of the handshake protocol on the Internet draft [6]. The draft does not include a precise list of requirements that must be satisfied by the communication channel created after the handshake protocol completes. Based on our interpretation of the informal discussion in Sections 1 and 5.5 of the Internet draft, we believe that the resulting channel can be considered "secure" if and only if the following properties hold:

- Let $Secret_C$ be the number that C considers the shared secret, and ***Secrets*** the number that S considers the shared secret. Then $Secret_C$ and Secrets must be identical.
- The secret shared between C and S is not in intruder's database of known message components.
- The parties agree on each other's identity and protocol completion status. Suppose that, the last message of the handshake protocol is from

Efficient Finite-State Analysis for Large Security Protocols

Vitaly Shmatikov Ulrich Stern

Computer Science Department
Stanford University
Stanford, CA 94305-9045

{shmat,uli}@cs.stanford.edu

Abstract

We describe two state reduction techniques for finite-state models of security protocols. ***The techniques exploit certain protocol*** properties ***that we have*** identified ***as characteristic*** of security protocols. ***We prove the soundness*** of ***the techniques by demonstrating that*** any ***violation*** of ***protocol*** invariants is preserved in ***the reduced state graph. In addition, we describe*** an ***optimization method*** for ***evaluating parameterized rule conditions, which*** are ***common*** in our ***models*** of ***security*** protocols. ***All*** three ***techniques have been implemented in the Murφ verifier.***

1 Introduction

Security protocols are becoming widely used and many new protocols are being proposed. Since security protocols are notoriously difficult to design, computer assistance in the design process is desirable. The existing verification methods are mainly based on either finite-state analysis, or computer-assisted proof. The two approaches are complementary. Unlike computer-assisted proof, finite-state analysis cannot guarantee correctness for a protocol of unbounded size (e.g., a protocol with a potentially unbounded number of participants). Finite-state analysis, however, requires much less human expertise and is fully automatic.

This work was supported in part by the Defense Advanced Research Projects Agency through contract DABT63-96-C-0097, Office of Naval Research grant N00014-97-1-0505, Multidisciplinary University Research Initiative "Semantic Consistency in Information Exchange," and the Hertz Foundation. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, ONR, or the US Government.

Finite-state analysis of security protocols begins with a high-level description of the behavior of the honest participants of the protocol. This protocol model must be augmented with a description of the possible actions of an intruder, and a precise statement of the desired properties of the protocol. The finite-state analysis tool then exhaustively enumerates all reachable states of the model, checking for each state whether it satisfies the desired correctness criteria. The main problem in this analysis is the very large number of reachable states for most protocols.

In this paper, we describe two techniques that reduce the number of reachable states and hence allow the analysis of larger protocols. We prove the techniques sound, i.e., we show that each protocol error that would have been discovered in the original state graph will still be discovered in the reduced state graph. The techniques are based on certain protocol properties that we have identified as characteristic of security protocols. We have implemented both techniques in the Murφ verification system [3] and have evaluated them on the SSL [4] and Kerberos [5] protocols.

The first technique is to let the intruder always intercept messages sent by the honest participants (instead of making such interception optional). This technique has resulted in a very large reduction in both the number of reachable states and execution time. While this technique has been used by several researchers [6, 1, e.g.], it has neither been proved sound, nor has its importance been demonstrated.

The second technique prevents the intruder from sending messages to honest participants in states where at least one of the honest participants is able to send a message. Intuitively, the technique makes the intruder more powerful since the intruder maximally increases its knowledge before forging and sending messages to honest participants; hence the technique should not

miss any attacks on the protocol. The technique typically saved a factor of two or more in the number of reachable states as well as execution time. It is interesting to note that that this technique is more powerful than partial-order techniques exploiting the independence of the honest participants.

In addition to the two state reduction techniques, we also describe a technique that reduces the execution time of Murcp, but not the number of reachable states. The technique is based on the following observations. The intruder model employed in Murφ is highly nondeterministic and thus gives rise to a large number of state transition rules. In every reachable state, the enabling conditions of all rules are evaluated. Evaluation can be sped up by partitioning the rules into sets with identical enabling conditions and evaluating the condition only once for each set. This technique typically increased the overall speed of Murcp by a factor of four.

2 Overview of Murcp

Murφ [2] is a protocol or, more generally, finite-state machine verification tool. It has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [3, 10, 11] and in the domain of security protocols [7, 8]. The purpose of finite-state analysis, commonly called "model checking," is to exhaustively search all execution sequences.

To verify a security protocol using Murcp, one has to model both the protocol and the intruder (or adversary) in the Murcp language and augment the resulting model with a specification of the desired properties. The Murcp system automatically checks, by explicit state enumeration, if every reachable state of the model satisfies the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The intruder is generally modeled to control the network and allowed the following actions: (1) overhear every message, remember all parts of each message (in a knowledge database), and decrypt ciphertext when it has the key, (2) intercept (delete) messages, and (3) generate messages using any combination of initial knowledge about the system and parts of overheard messages. We also assume that the intruder can masquerade as an honest participant in the system, capable of initiating communication with a truly honest participant, for example. We will refer to this intruder model as "mechanical intruder model" because of its simplicity.

The Murcp language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The ***state*** of the model consists of the values of all global variables. In ***a startstate*** statement, initial values are assigned to global variables. The transition from one state to another is performed by ***rules.*** Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e., the rule is enabled) and typically changes global variables, yielding a new state. Most Murφ models are nondeterministic since states typically allow execution of more than one rule. For example, in the model of a security protocol, the intruder usually has the nondeterministic choice of several messages to replay.

Murcp has no explicit notion of processes. Nevertheless a process can be implicitly modeled by a set of related rules. The ***parallel composition*** of two processes in Murcp is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of ***asynchronous, interleaving*** concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murcp language supports ***scalable*** models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a scaled-down (finite state) version of the protocol. Murcp can only guarantee correctness of the scaled-down version of the protocol, but not correctness of the general protocol. For example, the numbers of clients and servers in a security protocol are typically scalable and defined by constants.

The desired properties of a protocol can be specified in Murcp by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murφ prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

3 Properties of security protocols

In this section, we identify several characteristic properties of security protocols that we will use to develop state reduction techniques. These properties characterize every security protocol we have encountered so far, including, e.g., Kerberos [5], SSL [4], and Needham-Schroeder [9]. The properties are quite simple, yet recognizing them is useful both for better understanding of the protocols and for making finite-state analysis as efficient as possible within the basic framework of the mechanical intruder model.

3.1 Protocol invariants are monotonic

The invariants used to specify correctness of security protocols are typically of the forms "Intruder does not know X" or "If honest participant ***A*** reaches state S_1, then honest participant ***B*** must be in state S_2." Assume that an invariant of this form is invalid for a given state. Then increasing the intruder's knowledge set (which is part of the state) will not make the violated invariant valid. Hence we will call the protocol invariant ***monotonic*** with respect to the intruder's knowledge ***set.*** All protocol invariants we have encountered to date have been monotonic.

The implication of this property for state reduction is that we can safely rearrange the reachable state graph, possibly eliminating some states, as long as we can guarantee that for every state in the old graph, there exists a state in the new graph in which the state of all honest protocol participants is the same while the intruder's knowledge set is the same or larger. Because protocol invariants are monotonic, such state reductions are ***sound*** in the sense that any invariant that would have been violated in the old state graph will still be violated in the new graph.

We also assume that protocol invariants are defined in terms of the intruder's knowledge set and the states of the honest protocol participants. The invariants should not depend on the state of the network. (Since the network is assumed to be controlled by the intruder, invariants that depend on the state of the network can be rewritten to depend on the intruder's knowledge set.)

3.2 Intruder controls the network

It is traditionally assumed in security protocol analysis that the intruder exercises full control over the network, including the option to intercept any message. Instead of giving the intruder the option to intercept messages, we will assume that the intruder always intercepts. We model interception in Murφ by having the intruder remove the message from the "network" and store it in its database. The intruder can then replay the message to the intended recipient, or forge a similar-looking message.

Intuitively, the assumption that every message gets intercepted will not weaken the intruder and should hence be sound. In Section 5.1 below, this assumption is used to cut the transitions leading to redundant states that differ only in the contents of the intruder's database. The only state left is the one in which the database contains all information observable from the exchange of protocol messages up to that moment.

3.3 Honest protocol participants are independent

Honest protocol participants are fully independent from each other. The only means of communication between the participants is by sending messages on the network, which is assumed to be fully controlled by the intruder. Sending a message to another participant is thus equivalent to simply handing it to the intruder, hoping that the latter will not be able to extract any useful information from it and will replay it intact to the intended recipient.

As a consequence, an honest protocol participant has no way of knowing for sure what the current state of other participants is, since all information about the rest of the world arrives to it through a network fully controlled by the intruder. Actions of each honest participant (i.e., sending and receiving of messages) thus depend only on its own local state and not on the global state that comprises the states of all participants plus that of the intruder. In our formal representation of security protocols as state graphs, we will rely on this property to make all transition rules for honest participants local (see Section 4.2 below). We do not consider protocols with out-of-band communication as they are beyond the scope of our research with Murφ.

4 Protocols as state graphs

In this section, we define a formalism for describing finite-state machines associated with security protocols.

4.1 States

The global state of the system is represented by a vector:

$$s = [s_1, \ldots, s_N, e]$$

where N is the number of honest protocol participants, s_i is the local state of the protocol participant i, and e is the state of the intruder.

Instead of modeling the global network, we model a separate l-cell local network for each honest protocol participant. All messages intended for that participant are deposited in its local network as described in Section 4.2 below. The local state of an honest participant i is a pair

$$s_i = \langle v_i, m_i \rangle$$

where v_i is the vector of current values of i's local state variables, and m_i is the message currently in i's local network. It is possible that $m_i = \varepsilon$, representing the empty network.

The state of the intruder is simply the set of messages that the intruder has intercepted so far (assume that its initial knowledge is represented as an intercepted message also):

$$e = \{m_{e_1}, \ldots, m_{e_M}\}$$

The intruder's knowledge is obviously not limited to the intercepted messages. The intruder can split them into components, decrypt and encrypt fields, assemble new messages, etc. However, the full knowledge set can always be synthesized from the intercepted messages, since they are the only source of information available to the intruder. Therefore, our chosen representation for the intruder's state is sufficient to represent the intruder's knowledge. When necessary, we will refer to the set of messages that can be synthesized from the set of intercepted messages as $\mathrm{synth}(e)$. (Since operations like encryption and pairing can be applied infinitely many times in the synthesis, the intruder's full knowledge is generally infinite. In practice, one can extract finite limits on the numbers of times encryption and pairing have to be applied from the protocol definition, making the intruder's knowledge finite.)

4.2 Rules for honest participants

All transition rules between states have the following form in our formalism:

$$r_k^{(i|e)} = \text{if } c_k(s_i|e) \text{ then } s \rightarrow s'$$

where $c_k(s_i|e)$ is the condition of the rule (it depends on the local state s_i in case of an honest protocol participant, and the knowledge set e in case of the intruder), s is the original global state, and s' is the global state obtained as the result of the rule application.

We can assume without loss of generality that every transition rule for an honest protocol participant consists of reading a non-empty message off the local network, changing the local variables, and sending a non-empty message to another participant. If necessary, the protocol can be rewritten so as to avoid "hidden" transitions that change the state of a participant without visible activity on the network. The initial transition for each participant can be triggered by a special message deposited into its local network in the start state of the system, and the last transition can be rewritten so that it deposits another special message on the network. This ensures that every transition reads and writes into the network. Also, the l-cell capacity restriction on the local network is not essential, since we will eventually assume that every message is intercepted by the intruder immediately after it has been sent.

The transition rules for an honest protocol participant i are represented as follows:

$$r_k^{(i)} = \text{if } c_k(v_i, m_i) \text{ then } [\ldots \langle v_i, m_i \rangle \ldots \langle v_j, \varepsilon \rangle \ldots] \rightarrow [\ldots \langle v_i', \varepsilon \rangle \ldots \langle v_j, m_j \rangle \ldots]$$

Informally, if condition c_k evaluates to true given i's local state $s_i = \langle v_i, m_i \rangle$, then i reads message m_i off its local network, executes some code changing its local state variables from v_i to v_i', and sends message m_j to participant j by depositing it in j's local network. Note that the rule is local — its condition depends only on i's local state. We assume that honest participants are deterministic. In any state, there is no more than one rule enabled for each participant. However, it is possible that rules for several participants are enabled in the same state, resulting in nondeterminism.

4.3 Rules for the intruder

The transition rules for the intruder are global. The first set of rules describes the intruder intercepting a message intended for an honest participant:

$$r_{-i}^{(e)} = \text{if origin(m)} \neq \text{e then } [\ldots \langle v_i, \text{m} \rangle \ldots \text{e}] \rightarrow [\ldots \langle v_i, \varepsilon \rangle \ldots e \cup \{m\}]$$

The intruder first checks the origin of the message on i's local network, since we do not want the intruder to remove its own messages. If the message was generated by an honest participant, it is removed from the network and added to the intruder's database. Note that the local variables of the honest participant are not affected by this action.

Part II

Real-Time Systems

Hee-Hwan Kwak, Jin-Young Choi, *Insup* ***Lee, Anna Philippou, and Oleg Sokolsky:*** "Symbolic Schedulability Analysis of Real-time Systems", in Proceedings of the 19th IEEE Real-Time Systems Symposium — RTSS'98, IEEE Computer Society Press, to appear, Madrid, Spain, December 1998.

Full paper: `http: //www.cis.upenn.edu/~lee/tmp/98rtss.ps`

Max Kanovich, Mitsu *Okada,* ***and Andre Scedrov:*** "Specifying Real-Time Finite-State Systems in Linear Logic", in the Proceedings of the 2nd International Workshop on Constraint Programming for Time-Critical Applications and Multi-Agent Systems — COTIC'98, Electronic Notes in Theoretical Computer Science, to appear, Nice, France, September 1998.

Full paper: `file://www.cis.upenn.edu/pub/papers/scedrov/rtime6.ps.gz`

Anna Philippou, Oleg Sokolsky, *Insup* ***Lee, Rance Cleaveland, and Scott Smolka:*** "Probabilistic Resource Failure in Real-Time Process Algebra", in Proceedings of the 9th International Conference on Concurrency Theory — CONCUR'98, Springer-Verlag LNCS, to appear, Nice, France, September 1998.

Full paper: `http://www.cis.upenn.edu/~lee/tmp/98concur.ps`

Anna Philippou, Oleg Sokolsky, *Insup* ***Lee,*** Rance ***Cleaveland,*** and ***Scott Smolka:*** "Specifying Failures and Recoveries in PACSR", in the Proceedings of the Workshop on Probabilistic Methods in Verification, June 1998.

Full paper: `http: //www.cis.upenn.edu/~lee/tmp/98probmiv.ps`

Oleg ***Sokolsky, Insup Lee, and Hanêne Ben-Abdallah:*** "Specification and Analysis of Real-Time Systems with PARAGON", in Jeffrey J. P. Tsai editor, the Annals of Software Engineering, volume 7 (Real-Time Software Engineering), Baltzer Science Publishers, to appear. Submitted March 1998.

Full paper: `http://www.cis.upenn.edu/~lee/tmp/98ase.ps`

Oleg Sokolsky, Mohamed Younis, Insup Lee, Hee-Hwan Kwak, and Jeff Zhou: "Verification of the Redundancy Management System for Space Launch Vehicle: A Case Study", in the Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium — RTAS '98, IEEE Computer Society Press, to appear, Denver, CO, June 1998.

Full paper: http: `//www.cis.upenn.edu/~lee/tmp/98rtas.ps`

Symbolic Schedulability Analysis of Real-time Systems *

Hee-Hwan Kwak[1], Jin-Young Choi[2], Insup Lee[1],
Anna Philippou[1], and Oleg Sokolsky[3]

[1] Department of Computer and Information Science,University of Pennsylvania, USA.
{heekwak,annap}@saul.cis.upenn.edu, lee@cis.upenn.edu
[2] Department of Computer Science and Engineering, Korea University, Korea. choi@formal.korea.ac.kr
[3] Computer Command and Control Company, USA. sokolsky@cccc.com

Abstract

We propose a unifying method for analysis of scheduling problems in real-time systems. ***The method is based on ACSR-VP, a real-time process algebra with value-passing capabilities. We use ACSR-VP 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, a solution to which yields the values of the parameters that make the system schedulable. Equations are solved using integer programming or constraint logic*** programming. ***The paper presents specifications of two scheduling problems*** as examples.

*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

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 [6, 9]. 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-times tasks under a particular scheduling discipline can meet all of its timing constraints.

The pioneering work by Liu and Layland [15] 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 [10, 25, 23, 17, 24, 21, 16, 2]. 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 hard to perform. 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.

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) [13], 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 schedulability analysis problem [3], by assuming that all parameters are ground, i.e., constants. However, it is not possible to use the technique described in [3] to solve the general schedulability analysis problem with unknown timing parameters.

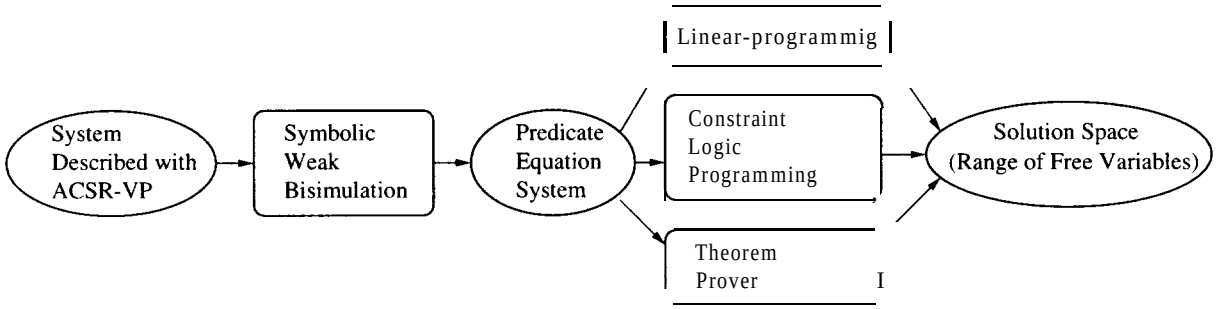

Figure 1: Overview

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 for unknown parameters, the system becomes schedulable.

The rest of the paper is organized as follows. Sections 2 and 3 overview the theory of the underlying formal method, ACSR-VP, and introduce symbolic bisimulation for ACSR-VP expressions. Section 4 gives specifications of two scheduling problems, namely the period assignment *problem* and the ***start-time assignment*** *problem.* Section 5 illustrates analysis of two instances of these problems. We conclude with a summary and an outline of future work in Section 6.

2 ACSR-VP

ACSR-VP extends the process algebra ACSR [13] 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 $BExpr \subset Expr$ 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 Z(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 $P(R \times Expr)$ 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 0 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 ovei $\mathcal{D}_R$. 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 *n,(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 π_r *(A) = 0* if $r \notin$ *p(A).* The entire domain of actions is

Specifying Real-Time Finite-State Systems in Linear Logic

Max I. Kanovich* Mitsuhiro Okada† Andre Scedrov‡

Abstract

Real-time finite-state systems may be specified in linear logic by means of linear implications between conjunctions of fixed finite length. In this setting, where time is treated as a dense linear ordering, safety properties may be expressed as certain provability problems. These provability problems are shown to be in PSPACE. They are solvable, with some guidance, by finite proof search in concurrent logic programming environments based on linear logic and acting as sort of model-checkers. One advantage of our approach is that either it provides unsafe runs or it actually establishes safety.

1 Introduction

There are a number of formalisms for expressing real-time processes, including [1, 6, 7, 3, 4, 5, 50, 44, 45, 38]. Many of these real-time formalisms are based on temporal logic or its variations [46, 38, 33] or on timed process algebras [14, 42, 43, 23, 12], or on Büchi automata [52, 3]. In some cases exact complexity-theoretic information is available, such as [51, 3, 5], while other formalisms are known to be undecidable. In this context, undecidability may arise *a priori* from the undecidability of traditional predicate logic with binary predicates, or in a more subtle way from so-called punctual temporal specifications, which are known to be capable of simulating the halting problem [5].

In this work we introduce a real-time specification formalism based on *linear logic* [19, 20, 48, 49]. A clear advantage of our approach is that it provides a common user with a very easy and transparent way of writing high-level specifications without having to be concerned with operational issues. Linear logic seems a natural choice for a *logical* specification formalism in this regard because of its intrinsic ability to reflect state transitions. Indeed, the most straightforward and naive way of writing very simple propositional logic formulas that correspond to the informal natural language descriptions of state transition systems is actually rigorous and correct in linear logic, while this way of writing specifications is incorrect in classical logic. This is discussed in detail below and in the railroad-crossing example in Section 2.

*max@kanovich. dnttm. rssi . ru Department of Theoretical and Applied Linguistics, Russian State University for the Humanities, Miusskaya 6, 125267 Moscow, Russia.

†mitsu@abelard. f let .mita.keio . ac. jp Department of Philosophy, Keio University, 2-15-45 Mita, Minato-ku, Tokyo 108, Japan. Partially supported by the Grant-in-Aid for Scientific Research (Ministry of Education, Science and Culture, Japan) and the Ogata-Jyosei Grant (Keio University).

‡scedrov@cis . upenn. edu http: //www. cis .upenn. edu/~scedrov Department of Mathematics, University of Pennsylvania, Philadelphia, PA 19104-6395 USA. Partially supported by DoD MURI "Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505, by NSF Grant CCR-9800785, and by an International Fellowship from the Japan Society for the Promotion of Science.

The best way to model, specify, and prove time-sensitive properties of real-time systems would be to use natural language. While this might be possible in the future, today it is customary to resort to various formal languages for this purpose. Among these, the formal language that has been most investigated and best understood is traditional predicate logic. In principle one could express various properties and requirements of real-time systems by means of formulas built up from certain basic, or atomic, predicates by traditional logical connectives and quantifiers. However, such a general approach in the framework of traditional predicate logic runs into difficulties, for example, the undecidability of predicate logic with binary predicates.

For purely qualitative time properties of real-time systems such ***as sometimes, always,*** *never*, it suffices to consider "time-closed" formulas where all time variables are bound by quantifiers. Such qualitative time properties can be handled within the ***temporal*** logic framework where all time variables are encapsulated by means of temporal-modal operators on the propositional level. There are a number of successful investigations in this line of research, for instance [46, 51]. However, one runs into difficulties with this approach in handling *quantitative* time properties such as ***"within B*** time ***units afterwards", "never for more than B time*** units", that refer to ***explicit*** time delays. In the case of such *quantitative* time constraints, in order to represent current states of a given real-time system temporal logic is to be equipped with first-order means so that ***time*** parameters cannot be handled but ***explicitly,*** beyond the propositional logical framework. There are other solutions, such as the ***temporal logic of actions*** TLA [33, 1], where real time is handled *explicitly* by introducing a variable to represent time. However, it is not easy to describe a decidable fragment of TLA suited for describing system requirements such ***as safety*** and *liveness.*

In this paper we introduce an approach that allows handling both qualitative and quantitative time aspects of real-time systems in purely logical terms, where the difficulties of being over-sophisticated and over-complicated are obviated within the framework of ***monadic Horn fragment*** of linear logic in the sense of [28]. This simple fragment of linear logic can be communicated to the common users without requiring any sophistication in logic. Let us describe the main idea of our approach. For real-time systems with their peculiar time, one of the basic primitive relations one deals with is of the form

$$P(e, t) \equiv \text{"an event e happens in the system at moment f'.}$$

In order to circumvent the difficulties caused by binary predicates (which, for example, usually lead to undecidability of the system), one may split $P(e, t)$ into two *unary* predicates: a "timeless predicate" Q(e) that means "event e happens in the system" and the unique "time predicate" Time(t) that means "time is t (on the global clock)". That is,

$$P(e, t) \approx (\text{Q(e) and Time(t)}).$$

Suppose that a given action is performed in such a way that a certain event e_1 at moment t_1 is followed by another event e_2 at moment t_2 (as a delayed effect). A naive way of formalizing this action is by a "Horn axiom" of the form $P(e_1, t_1)$ **implies** $P(e_2, t_2)$. Following our unarization procedure, this axiom is supposed to be encoded as:

$$(\mathtt{Q}(\mathtt{e}_1) \textbf{ and } \mathtt{Time}(\mathtt{t}_1)) \textbf{ implies } (\mathtt{Q}(\mathtt{e}_2) \textbf{ and } \mathtt{Time}(\mathtt{t}_2)).$$

However, such a straightforward reduction to unary predicates requires certain precautions related to the exact meaning of the connectives **and** and **implies.** In particular, the traditional understanding of and and **implies** as boolean connectives $\wedge$ and $\Rightarrow$, respectively, yields unintended consequences such as

$$(\mathtt{Q}(\mathtt{e}_1) \wedge \mathtt{Time}(\mathtt{t}_1)) \Rightarrow (\mathtt{Q}(\mathtt{e}_1) \wedge \mathtt{Q}(\mathtt{e}_2) \wedge \mathtt{Time}(\mathtt{t}_1) \wedge \mathtt{Time}(\mathtt{t}_2)),$$

that is, $P(e_1, t_1) \Rightarrow (P(e_1, t_1) \wedge P(e_2, t_2))$, and furthermore

$$(\mathtt{Q}(\mathtt{e}_1)\ \mathbf{A}\ \mathtt{Time}(\mathtt{t}_1)) \Rightarrow (\mathtt{Q}(\mathtt{e}_1) \wedge \mathtt{Time}(\mathtt{t}_2)),$$

that is, $P(e_1, t_1) \Rightarrow P(e_1, t_2)$.

The main reason behind these problems is that our **and** is intended to represent only the "concurrent coexistence", while the traditional conjunction A can behave in many different ways. This is one reason why as a possible logical setting we propose linear logic, a resource-sensitive refinement of classical logic [19, 20, 48, 49], where the traditional conjunction A is split into two connectives: $\otimes$ (tensor) and & (with), and the traditional implication $\Rightarrow$ is refined as ***linear implication*** $\multimap$. Revealing the ***"concurrent coexistence"*** nature of $\otimes$, we encode the basic binary relation $P(e, t)$ by a linear logic formula of the form Q(e) $\otimes$ Time(t). Accordingly, the action discussed above will be specified by a linear logic implication of the form

$$(\mathtt{Q}(\mathtt{e}_1) \otimes \mathtt{Time}(\mathtt{t}_1)) \multimap (\mathtt{Q}(\mathtt{e}_2) \otimes \mathtt{Time}(\mathtt{t}_2)).$$

It is remarkable that in linear logic this formula does not yield the undesired formula

$$(\mathtt{Q}(\mathtt{e}_1) \otimes \mathtt{Time}(\mathtt{t}_1)) \multimap (\mathtt{Q}(\mathtt{e}_1) \otimes \mathtt{Q}(\mathtt{e}_2) \otimes \mathtt{Time}(\mathtt{t}_1) \otimes \mathtt{Time}(\mathtt{t}_2)),$$

nor the undesired formula

$$(\mathtt{Q}(\mathtt{e}_1) \otimes \mathtt{Time}(\mathtt{t}_1)) \multimap (\mathtt{Q}(\mathtt{e}_1) \otimes \mathtt{Time}(\mathtt{t}_2)).$$

Concrete examples of this phenomenon are discussed in the railroad-crossing example in Section 2.

Important system properties such as ***safety*** are represented in our approach as certain PSPACE decision properties related to ***provability*** in linear logic. In terms of complexity, this indicates a good fit with the automata-theoretic approach [3, 4] and its PSPACE-complete problem of emptiness of the language associated to an automaton, in contrast with the EXPSPACE-complete properties related to *satisfiability* in metric interval temporal logic [5]. By way of comparison between our setting and the automata-theoretic approach, let us emphasize that one of the central concepts used in verification is ***reachability,*** in the sense that safety is seen as unreachability. Our approach provides a simple and direct correspondence between reachability and the traditional logical concept of provability. In contrast, the traditional concept emphasized in the automata-theoretic approach is the language emptiness problem, while reachability is treated there only as a derived, subsidiary notion. Moreover, the way reachability is derived from language emptiness in the automata-theoretic approach involves nontrivial technical operations such as language intersection and complementation. The exact nature of a relationship between our approach and the automata-theoretic approach remains to be determined.

Let us note that the method of proof of our main complexity result shows that, aside from complexity bounds, decision problems that involve temporal constraints may be dealt with by running a finite ***proof*** search, with some guidance, in the available concurrent logic programming environments based on linear logic [9, 10, 24, 25, 41, 30, 31, 32, 13] or in the environments supporting multiset rewriting [17, 22] or concurrent rewriting [40, 44], either of which would in this case act as sort of model-checkers. Indeed, our current work may be seen as a first step toward a larger issue of ***proof-bused state exploration*** in contrast to model-checking, which is model-based. One advantage of our approach is that it incorporates a decision procedure, so that either it provides unsafe runs or it actually establishes safety.

Technically, our framework may be seen as a combination of local transitions and ***global,*** quantitative time correlations. In our framework transitions are instantaneous but events may have duration.

Regarding the transitions, our framework is a refinement of the work in [11, 21, 18, 39, 15], which established a direct relationship between Petri nets and linear logic axiomatizations using conjunctive formulas. Here we consider only conjunctions of fixed finite length. In linear logic this restriction suffices for a faithful simulation of finite state transitions.

We extend this underlying framework to real-time systems by using global constraints formulated by means of alarms (timers, time guards.) Our use of these devices is generally motivated by "an old-fashioned approach" in [1], although our actual technical treatment of the timers is somewhat different. Let us illustrate our combination of local transitions and global time constraints in more detail on the standard railroad-crossing example.

2 Example: Railroad-Crossing Controller

The railroad-crossing system we consider consists of a train, a signal, and a gate. The train goes from being safe to approaching, then to crossing, and then back to being safe. The signal may be set to either raise or lower. The gate has four options: up, down, moving-up, or moving-down.

The controller senses when the train starts approaching and sets the signal to lower within D time units. When the signal is set to lower, then the gate starts moving-down within G time units. Once the gate starts moving-down, it is down within L time units. When the train is safe, the signal is set to raise, and in turn, the gate starts moving-up, and is then up. For the purposes of this simple example, no time bounds are placed on this suite. In addition to that, the train is supposed to spend at least B time units going from safe to crossing.

The main ***safety property*** of the system is that when the train is crossing, then the gate is down. A common assumption is that $B > D + G + L$.

2.1 Specifying Timeless Transitions

In order to let the reader develop some feel for the way linear logic operates, let us first discuss in detail linear logic specifications of timeless transitions. Because of the resource-sensitive nature of linear logic, the most naive way of writing logical formulas corresponding to the informal English description above is actually rigorous and correct. For instance, changing the signal from lower to raise when the train is safe is specified by the formula:

$$(\mathtt{Tr(safe)} \otimes \mathrm{Sig(low)}) \multimap (\mathtt{Tr(safe)} \otimes \mathrm{Sig(raise)}), \qquad (1)$$

where $\otimes$ (tensor) is a linear logic version of conjunction and $\multimap$ is ***linear implication.*** The meaning of a linear implication $A \multimap B$ is not simply that A implies B but that A ***is consumed*** or ***spent*** and that B is produced. This is reflected in the linear logic rules of inference. For instance, a linear logic counterpart $A \multimap (A \otimes A)$ of the traditional propositional tautology $A \Rightarrow (A \wedge A)$ is not provable in linear logic. This expressive ability of linear logic to distinguish between one and ***two*** occurrences of a formula reflects the common sense that \$1 cannot be spent to produce a \$1 and another \$1. In our situation this expressive ability makes it possible for the linear logic specification (1) to stipulate that the signal changes from lower to raise. A similar formula in classical or in intuitionistic logic

$$(\mathtt{Tr(safe)} \wedge \mathtt{Sig(low)}) \Rightarrow (\mathrm{Tr(safe)} \wedge \mathrm{Sig(raise)})$$

is incorrect in this regard, because the tautology $A \Rightarrow (A \wedge A)$ allows us to infer

$$(\mathtt{Tr(safe)} \wedge \mathtt{Sig(low)}) \Rightarrow (\mathtt{Tr(safe)} \wedge \mathtt{Sig(low)} \wedge \mathtt{Sig(raise)}),$$

Probabilistic Resource Failure in Real-Time Process Algebra*

Anna Philippou[1], Rance Cleaveland[2], Insup Lee[1],
Scott Smolka[3], and Oleg Sokolsky[4]

[1] University of Pennsylvania, USA. {annap, lee}@saul.cis.upenn.edu
*University of North Carolina, USA. rance@eos.ncsu.edu
[3]SUNY at Stony Brook, USA. sas@cs.sunysb.edu
[4]Computer Command and Control Company, USA. sokolsky@cccc.com

Abstract. PACSR, a probabilistic extension of the real-time process algebra ACSR, is presented. The extension is built upon a novel treatment of the notion of a *resource*. In ACSR, resources are used to model contention in accessing physical devices. Here, resources are invested with the ability to *fail* and are associated with a probability of failure. The resulting formalism allows one to perform probabilistic analysis of real-time system specifications in the presence of resource failures. A probabilistic variant of Hennessy-Milner logic with *until* is presented. The logic features an *until* operator which is parameterized by both a probabilistic constraint and a regular expression over observable actions. This style of parameterization allows the application of probabilistic constraints to complex execution fragments. A model-checking algorithm for the proposed logic is also given. Finally, PACSR and the logic are illustrated with a telecommunications example.

1 Introduct ion

A common high-level view of a distributed real-time system is that the components of the system compete for access to shared resources, communicating with each other as necessary. To capture this view explicitly in formal specifications, a real-time process algebra ACSR [17] has been developed. ACSR represents a real-time system as a collection of concurrent processes. Each process can engage in two kinds of activities: communication with other processes by means of instantaneous ***events*** and computation by means of timed ***actions.*** Executing an action requires access to a set of resources and takes a non-zero amount of time measured by an implicit global clock. Resources are serially reusable, and access to them is governed by priorities. A process that attempts to access a resource currently in use by a higher-priority process is blocked from proceeding.

The notion of a resource, which is already important in the specification of real-time systems, additionally provides a convenient abstraction mechanism for

* This work was supported in part by grants AFOSR F49620-95-1-0508, ARO DAAH04-95-1-0092, NSF CCR-9415346, NSF CCR-9619910, and ONR N00014-97-l-0505 (MURI).

probabilistic aspects of systems behavior. A major source of behavioral variation in a process is failure of physical devices, such as processors, memory units, and communication links. These are exactly the type of objects that are captured as resources in ACSR specifications. Therefore, it is natural to use resources as a means of exploring the impact of failures on a system's performance.

In this paper, we present PACSR, a process algebra that extends the resource model of ACSR with the ability to reason about resource failures. Each resource used by the system is associated with a probability of failure. If a process attempts to access a failed resource, it is blocked. Resource failures are assumed to be independent. Then, for each execution step. that requires access to a set of resources, we can compute the probability of being able to take the step. This approach allows us to reason quantitatively about a system's behavior.

Previous work on extending process algebra with probability information (discussed below) typically associates probabilities with process terms. An advantage of associating probabilities with resources, rather than with process terms, is that the specification of a process does not involve probabilities directly. In particular, a specification simply refers to the resources required by a process. Failure probabilities of individual resources are defined separately and are used only during analysis. This makes the specification simpler and ensures a more systematic way of applying probabilistic information. In addition, this approach allows one to explore the impact of changing probabilities of failures on the overall behavior, without changing the specification.

We are also interested in being able to specify and verify high-level requirements for a PACSR specification. Temporal logics are commonly used to express such high-level requirements. In the probabilistic setting, the requirements usually include probabilistic criteria that apply to large fragments of the system's execution. We present a simple temporal logic suitable for expressing properties of PACSR expressions. As is common with probabilistic extensions of temporal logics, we associate probabilistic constraints with temporal operators. The novel feature of the logic is that we allow temporal operators to be parameterized with regular expressions over the set of observable actions. Such parameterization allows us to apply probabilistic constraints to complex execution fragments.

For example, consider a communication protocol in which a sender inquires about the readiness of a receiver, obtains an acknowledgement, and sends data. A reasonable requirement for the system would be that this exchange happens with a certain probability. To express this property, one usually needs two nested temporal ***until*** operators. Since probabilistic constraints are associated with temporal operators, the single constraint has to be artificially split in two to apply to each of the operators. With the proposed extension, we need only one temporal operator, and the property is expressed naturally. A model-checking algorithm for the logic, suitable for finite-state PACSR specifications, is also given.

In terms of related work, a number of process algebras have been proposed that extend process terms with probability information, including [12, 21, 2, 10, 16,201. The approach of [12] is particularly relevant as it also adds probability to a real-time process algebra. It does not, however, consider the notions of

resource and resource probability, nor use priorities to control communication and resource access. In [18], an automata-based formalism that combines the notions of real-time and probabilities is presented. It employs a different notion of time in that transitions can have variable durations. Also, probabilities are associated with instantaneous events.

Since a PACSR specification typically consists of several parallel processes, concurrent events in these processes are the source of non-deterministic behavior, which cannot be resolved through probabilities. To provide for both probabilistic and non-deterministic behavior, the semantics of PACSR processes are given via ***labeled concurrent Markov chains*** [22]. This model has also been employed in [12], and variations of it appeared in [18, 6].

Regarding previous work on model checking for probabilistic systems, a closely related approach involves associating a probability threshold with the *until* operator of the temporal logic CTL [7]. For example, see [13, 4, 6, 14]. We find that this approach can become problematic when expressing properties that require multiple, nested *until*s. Our proposed extension of the ***until*** operator, which uses regular expressions and probability, serves to alleviate this deficiency.

The rest of the paper is organized as follows: the next section presents the syntax of PACSR and its semantics is given in Section 3. Section 4 discusses the temporal logic and the model-checking algorithm. In Section 5, we present an application of PACSR for the analysis of a probabilistic telecommunications system. We conclude with some final remarks and discussion of future work.

2 The Syntax of PACSR

2.1 Resource Probabilities and Actions

PACSR (Probabilistic ACSR) extends the process algebra ACSR by associating with each resource a probability. This probability captures the rate at which the resource may fail. PACSR also has two types of actions: instantaneous events and timed actions, the latter of which specifies access to a (possibly empty) set of resources. We discuss these three concepts below.

Instantaneous *events.* PACSR instantaneous actions are called ***events.*** Events provide the basic synchronization primitives in the process algebra. An event is denoted as a pair (a, p), where ***a*** is the ***label*** of the event and p, a natural number, is the *priority.* Labels are drawn from the set L = C $\cup$ $\overline{\mathcal{L}}$ $\cup$ $\{\tau\}$, where if a is a given label, $\overline{a}$ is its inverse label. The special label τ arises when two events with inverse labels are executed concurrently. We let a, ***b*** range over labels. Further, we use $\mathcal{D}_E$ to denote the domain of events.

Timed ***actions.*** We assume that a system contains a finite set of serially reusable resources drawn from the set ***Res.*** We also consider set ***Res*** that contains, for each $r \in$ ***Res,*** an element $\overline{r}$, representing the ***failed*** resource ***r.*** We write R for ***Res*** $\cup$ ***Res.*** An action that consumes one tick of time is drawn from the domain $\mathcal{P}(\mathsf{R} \times \mathbb{N})$ with the restriction that each resource is represented at most once. For

example the singleton action $\{(r, p)\}$ denotes the use of some resource $r \in Res$ at priority level p. Such an action cannot happen if r has failed. On the other hand, action $\{(\overline{r}, q)\}$ takes place with priority q given that resource r has failed. This construct is useful for specifying recovery from failures. The action 0 represents idling for one unit of time, since no resource is consumed.

We let $\mathcal{D}_R$ denote the domain of timed actions and we let A, B range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of the resources used by action A; for example $\rho(\{(r_1, p_1), (\overline{r_2}, p_2)\}) = \{r_1, \overline{r_2}\}$.

Resource Probabilities In PACSR we associate each resource with a probability specifying the rate at which the resource may fail. In particular, for all r $\in$ ***Res*** we denote by p(r) $\in$ [0, 1] the probability of resource r being up, while $\mathsf{p}(\overline{r}) = 1 - \mathsf{p}(r)$ denotes the probability of r failing. Thus, the behavior of a resource-consuming process has certain probabilistic aspects to it which are reflected in the operational semantics of PACSR. For example, consider process $\{(cpu, 1)) : \mathrm{NIL}$, where resource cpu has probability of failure $1/3$, i.e. $\mathsf{p}(cpu) = 2/3$. Then with probability $2/3$, resource cpu is available and thus the process may consume it and become inactive, while with probability $1/3$ the resource fails and the process deadlocks. This is discussed in detail in Section 3.

2.2 Processes

We let P, Q range over PACSR processes and we assume a set of process constants each with an associated definition of the kind $X \stackrel{\text{def}}{=} P$. The following grammar describes the syntax of PACSR processes.

$$P ::= \mathrm{NIL} \mid (a, n).\ P \mid A : P \mid P + P \mid P \| P \mid P \ \Delta^{a}_{t}\ (P, P, P) \mid P \backslash F \mid [P]_I \mid P \backslash\backslash I \mid rec\ X.P \mid X$$

The process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The first, $(a, n).\ P$, executes the instantaneous event (a, n) and proceeds to P. When it is not relevant for the discussion, we omit the priority of an event in a process. The second, $A : P$, executes a resource-consuming action during the first time unit and proceeds to process 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 events, and they synchronize on timed actions.

The scope construct, P A", (Q, R, S), binds the process P by a temporal scope and incorporates the notions of timeout and interrupts. We call t the ***time bound,*** where $t \in \mathbb{N} \cup \{\mathrm{co}\}$ and require that P may execute for a maximum of t time units. The scope may be exited in one of three ways: First, if P terminates successfully within t time-units by executing an event labeled $\overline{a}$, where a $\in$ L, then control is delegated to Q, the success-handler. On the other hand, if P fails to terminate within time t then control proceeds to R. Finally, throughout execution of this process construct, P may be interrupted by process S. In $P \backslash F$,

Specifying Failures and Recoveries in PACSR *

Anna Philippou[1], Oleg Sokolsky[2], Insup Lee[1],
Rance Cleaveland[3], and Scott Smolka[4]

[1]University of Pennsylvania, USA. {annap,lee}@saul.cis.upenn.edu
[2]Computer Command and Control Company, USA. sokolsky@cccc.com
[3]University of North Carolina, USA. rance@eos.ncsu.edu
[4]SUNY at Stony Brook, USA. sas@cs.sunysb.edu

Abstract

The paper presents PACSR, a probabilistic extension of a real-time process algebra ACSR. The extension is built upon a novel treatment of the notion of a ***resource.*** In ACSR, resources are used to model contention in accessing physical devices such as processors, memory modules, and communication links, or any other reusable resource of limited capacity. Here, we invest resources with an ability to *fail* and associate, with every resource, a probability of its failure. The resulting formalism allows us to perform probabilistic analysis of real-time system specifications in the presence of resource failures. An attractive feature of PACSR is the ability to express failure-recovery actions easily.

We perform probabilistic reachability analysis for PACSR specifications that allows us to compute the probability of occurrence of an undesirable event. We illustrate PACSR specification and analysis by means of a telecommunications example.

1 Introduction

Process algebras such as CCS [15] have proved to be effective for specification and analysis of distributed systems. Numerous real-time [22, 16, 12] and probabilistic [20] extensions of process algebras exist. We propose an approach that allows one to perform probabilistic analysis for real-time systems.

A common high-level view of a distributed real-time system is that its components compete for access to shared resources, communicating with each other as necessary. To capture this view explicitly in formal specifications, a real-time process algebra ACSR [14] has been developed. ACSR represents a real-time system as a collection of concurrent processes. Each process can engage in two kinds of activities: communication with other processes by means of instantaneous events and computation by means of timed actions. Executing an action requires access to a set of resources and takes a non-zero amount of time measured by an implicit global clock. Resources are serially reusable, and access to them is governed by priorities. A process that attempts to access a resource currently in use by a higher-priority process is blocked from proceeding.

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

The notion of a resource, which is important in specification of real-time systems, is even more critical to capture the probabilistic aspects of real-time systems behavior. A major source of behavioral variations in a process is failure of physical devices, such as processors, memory units, and communication links, that the process utilizes during its execution. These are exactly the type of objects that are captured as resources in ACSR specifications. Therefore, it is natural to use resources as a means of exploring the impact of failures on a system's performance.

In this paper, we present PACSR, a process algebra that extends the resource model of ACSR with the ability to reason about resource failures. With each resource used by the system, we associate a fixed probability of failure. If a process attempts to access a failed resource, it is blocked. Resource failures are assumed to be independent. Then, for each execution step that requires access to a set of resources, we can compute the probability of being able to take the step. This approach allows us to reason quantitatively about a system's behavior.

An advantage of associating probabilities with resources, rather than with process terms, is that the specification of a process does not involve probabilities directly. In particular, a specification simply refers to the resources required by a process. Failure probabilities of individual resources are defined separately and are used only during analysis. This makes the specification simpler and ensures a more systematic way of applying probabilistic information. In addition, this approach allows one to explore the impact of changing probabilities of failures on the overall behavior, without changing the specification.

A related approach that combines probabilistic specification with the notion of time is presented in [10]. The main distinguishing features of PACSR are the notion of resources and their use to capture probabilistic data, and the use of priorities to control communication and resource access.

A synchronous probabilistic process algebra WCCS is presented in [19]. There, each choice is assigned a weight. Weights are treated as priorities of the corresponding computation path. Furthermore, weights provide for probabilistic analysis of the specification. Time is measured in WCCS by counting the number of actions performed by a process, and no high-level temporal constructs such as timeouts are provided.

In [17], an automata-based formalism that combines the notions of real-time and probabilities is presented. It employs a different notion of time in that transitions can have variable durations. Also, probabilities are associated with instantaneous events.

Since a PACSR specification typically consists of several parallel processes, concurrent events in these processes are the source of non-deterministic behavior, which cannot be resolved through probabilities. To provide for both probabilistic and non-deterministic behavior, semantics of PACSR processes are given via labeled concurrent *Markov* chains [21]. This model has also been employed in [10], and variations of it appeared in [17, 5].

We employ probabilistic reachability as means of analysis of PACSR specifications. The method allows us to perform quantitative analysis of safety properties by computing the probability of observing an undesirable event. Another popular method of analysis is probabilistic model checking [11, 3, 5].

The rest of the paper is organized as follows: In the next section we present the syntax of PACSR and then we proceed with its semantics in Section 3. In Section 4, we discuss probabilistic reachability for PACSR terms. In Section 5, we present an application of PACSR for the analysis of a probabilistic system. We conclude with some final remarks

and discussion of future work.

2 The Syntax of PACSR

2.1 Actions

PACSR extends the process algebra ACSR with probability by enriching the notion of resource, associating each resource with a probability. This probability captures the rate at which the resource may fail. PACSR has three types of actions: timed actions, events and probabilistic actions. We discuss these below:

Timed actions. We assume that a system contains a finite set of serially-reusable resources drawn from the set Res. We also consider set Res that contains, for each $r \in Res$, an element $\overline{r}$, representing the ***failed*** resource r. Finally, we write R for Res ∪ ***Res.*** An action that consumes one tick of time is drawn from the domain $\boldsymbol{P}(\mathsf{R} \times \boldsymbol{N})$ with the restriction that each resource is represented at most once. For example the singleton action $\{(r,p)\}$ denotes the use of some resource $r \in$ ***Res*** at priority level p. Such action cannot happen if r has failed. On the other hand, action $\{(\overline{r},q)\}$ takes place with priority ***q*** given that resource r has failed. This construct is useful for specifying recovery from failures. 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 and we let ***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 $\rho(\{(r_1,p_1),(\overline{r_2},p_2)\}) = \{r_1,\overline{r_2}\}$.

Instantaneous events. PACSR instantaneous actions are called events. Events provide the basic synchronization primitives in the process algebra. An event is denoted as a pair (a,p), where a is the ***label*** of the event and p is the priority. Labels are drawn from the set $\mathsf{L} = \mathcal{L} \cup \overline{\mathcal{L}} \cup \{\tau\}$, where if a is a given label, $\overline{a}$ is its inverse label. The special label τ, arises when two events with inverse labels are executed concurrently. We let a, b, range over labels. Further, we use $\mathcal{D}_E$ to range over the domain of events.

Probabilistic actions. As mentioned earlier, in PACSR we associate each resource, with a probability capturing the rate at which the resource may fail. In particular, for all $r \in$ ***Res*** we denote by p(r) ∈ [0, 1] the probability of resource r being up, while $\mathsf{p}(\overline{r}) = 1 - \mathsf{p}(r)$ denotes the probability of r failing. Thus, the behavior of a resource-consuming process has probabilistic aspects that are captured by probabilistic actions. For example, consider process $\{(cpu, 1))$: NIL where resource *cpu* has probability of failure $1/3$, i.e. $\mathsf{p}(cpu) = 2/3$. Then with probability $2/3$, resource cpu is available and thus the process may consume it and become inactive, while with probability $1/3$ the resource may fail, in which case the process deadlocks. This will be discussed in more detail in Section 3.

2.2 Processes

We let P, Q range over PACSR processes and we assume a set of process constants each with an associated definition of the kind $\mathrm{X} \stackrel{\text{def}}{=} \mathrm{P}$. The following grammar describes the

syntax of PACSR processes.

$$P ::= \text{NIL} \mid A:P \mid (a,n).P \mid P+P \mid P\|P \mid P\ \Delta_t^a\ (P, P, P) \mid P\backslash F \mid [P]_I \mid P\backslash\backslash I \mid rec\ X.P \mid X$$

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 $(a,n).P$, executes the instantaneous event (a,n) and proceeds to P. Sometimes, when it is not relevant for the discussion, we omit the priority of an event in a process. 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. The scope construct, $P\ \Delta_t^a$ (Q, R, S), binds the process P by a temporal scope and incorporates the notions of timeout and interrupts. We call t the ***time bound,*** where $t \in \mathbb{N} \cup \{\infty\}$ and require that P may execute for a maximum of t time units. The scope may be exited in one of three ways: First, if P terminates successfully within the time bound t by executing an event labeled $\overline{a}$, where a $\in$ L, then control is delegated to process Q, the success-handler. On the other hand, if P fails to terminate within time t then control proceeds to R. Finally, throughout execution of this process construct, P may be interrupted by process S. 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 $[P]_I$, $I \subseteq \mathcal{R}$, produces a process that reserves the use of resources in I for itself, extending every action A in P with resources in $I - p(A)$ at priority 0. $P\backslash\backslash I$ hides the identity of resources in I so that they are not visible on the interface with the environment. Finally, the process $rec\ X.P$ denotes standard recursion. We write Proc for the set of PACSR processes.

The operator $P\backslash\backslash I$ binds all free occurrences of the resources of I in P. This binder gives rise to the sets of *free* and *bound resources* of a process P. In what follows, we work up to a-conversion on resources so as to avoid tedious side conditions. In this way, bound resources in a process are assumed to be different from each other and from the other free resources, and o-equivalent processes are assumed to have the same transitions.

Note that the syntax of PACSR processes is the same as that of ACSR. The only extension concerns the appearance of failed resources in timed actions. This allows us to perform probabilistic analysis of existing ACSR specifications without any modifications, as well as use non-probabilistic analysis of PACSR processes (without failure recovery actions).

The informal account of behavior just given is made precise via a family of rules that define the labeled transition relations $\longrightarrow_\pi$ and $\longmapsto$ on processes. This is presented in the next section. First we have some useful definitions.

The function $\mathsf{imr}(P)$, defined inductively below, associates each PACSR process with

Specification and Analysis of Real-Time Systems with PARAGON*

Oleg Sokolsky
Computer Command and Control Company
sokolsky@cccc.com

Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, U.S.A.
lee@central.cis.upenn.edu

Hanêne Ben-Abdallah
Dkpartement d'Informatique
FSEG
Université de Sfax
Sfax, Tunisia
hanene@saul.cis.upenn.edu

August 6, 1998

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 a real-time system. For analysis, it offers automatic analysis based on state space exploration as well as user-directed simulation. Our experience of using PARAGON on 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 allow users to specify their 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

*This research was supported in part by NSF CCR-9415346, NSF CCR-9619910, AFOSR F49620-95-1-0508, AFOSR F49620-96-1-0204, ONR N00014-97-1-0505 (MURI),

developed. They include state machines, Petri nets, logics, temporal logics, process algebra and timed automata; the summary of existing approaches and directions for future research can be found in [14, 17]. As formal methods become more mature and their benefits for development of large system can be clearly demonstrated, formal methods 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 [16], Spin [29], SMV [37]. Analysis of real-time systems is supported by COSPAN [24], Kronos [18], and Uppaal [S].

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 infinite, or finite but unmanageable, state space of the system's specification. Users of each formalism and supporting tools employees 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, these 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 the toolset, called 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 aimed at reduction of the state space of specifications, commonly employed by PARAGON.

PARAGON is based on process algebra ACSR [33] and related formalisms. Process algebras, such as CCS [38], CSP [28] and ACP [9], have been developed 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 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 indicates 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.* [33], is a timed process algebra which can be regarded as an extension of CCS. It enriches the set of operators, introducing constructs to capture such common real-time design notions such as resource sharing and 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, called CCSR [22], 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 [11]. To promote the use of ACSR in the specification and analysis of real-time systems, we have implemented a tool called VERSA [13]. PARAGON is a toolset that extends the capability of VERSA by providing graphical user interface, graphical specification language and simulation, as well as modifying it to handle larger specifications.

The paper is organized as follows. Section 2 presents the paradigm of Communicating Shared Resources, which forms the basis for PARAGON analysis. Section 3 gives an overview of the PARAGON toolset. Section 4 describes the specification and analysis methodology of PARAGON using a real-life example. We also detail heuristics that enabled us to successfully analyze this example. Section 5 gives an overview of related work. We conclude in Section 6 with a summary of our results and directions for future research.

2 Overview of the Formalism

The specification paradigm of Communicating Shared Resources (CSR) [21] is the basis for several process-algebraic formalisms. Among these formalisms are the real-time process algebra ACSR (the Algebra of Communicating Shared Resources) [11, 33] and a visual specification language GCSR (the Graphical CSR) [4, 6]. The two languages have compatible semantics and can be intermixed in a large specification.

The CSR paradigm is based on the view that a real-time system consists of a set of communicating components called processes. Processes compete for access to a finite set of serially shared resources and synchronize with one another through communication channels.

Further, parameterized specifications allow users to represent data manipulation and value passing between processes.

The use of shared resources by processes is represented by timed *actions,* and synchronization is supported via instantaneous *events.* The execution of an action is assumed to utilize a set of resources during a nonzero amount of time, measured by an implicit global clock. The execution of an action is subject to availability of resources it uses, and contention for resources is arbitrated according to priorities of competing actions. In addition, to ensure uniform progress of time, processes execute actions synchronously. Time can be either dense or discrete; in this paper, we consider only discrete time semantics to simplify the presentation. With discrete time, duration of an action is one tick of the global clock. Each action is represented by a set of resources needed for the action, each with an access priority. Example of an action is $\{(cpu, 2), (\text{sensor}, 1)\}$, which is executed only if resources *cpu* and *sensor* are not in use by a higher-priority process.

Unlike an action, the execution of an event is instantaneous and does not require any resources. Processes execute events asynchronously except when two processes synchronize through matching event names, i.e., channels. Two events match if one is an input and the other an output on the same-name channel. Matching pairs of events are denoted a and $\bar{a}$ in ACSR, respectively, and a? and a! in GCSR. Contention for channels is also resolved according to priorities of events. An input event in channel a with priority 1 is denoted as $(a, 1)$.

We next present the syntax and informal semantics of ACSR and GCSR, two languages that implement the CSR specification paradigm. Formal semantics for ACSR and GCSR in the form of structured operational semantics (SOS) rules may be found in [33] and [7], respectively.

2.1 Syntax and Semantics of ACSR

ACSR provides a set of operators that are similar to the common set of operators found in other process algebras: *prefix* for sequencing of actions and events; choice for choosing between alternatives; *parallel* for composing two processes to run in parallel; *restriction* and *hiding* for abstracting communication details or resource names; and recursion for describing infinite processes. As a real-time formalism, ACSR supports a variety of operators that deal with time. They allow one to delay execution for t time units, to *timeout* while waiting for some actions to occur, and to *bound* the time it takes to execute a sequence of actions. In addition, ACSR provides two operators, *interrupt* and *exception,* that are extremely useful in modeling real-time systems but are not present in other real-time process algebras. The interrupt operator makes it easy to specify reaction to asynchronous actions or events. The exception operator allows an exception to be raised any place inside a process and handled by an exception handling process.

ACSR provides for parameterization of a process specification by an index set. to support efficient representation families of similarly defined processes. There are two kinds of variables in ACSR specifications, process variables and *index variables.* Process variables represent ACSR terms and index variables range over elements of some index set and are

Verification of the Redundancy Management System for Space Launch Vehicle * A Case Study

Oleg Sokolsky*, Mohamed Younis†, Insup Lee:, Hee-Hwan Kwak‡ and Jeff Zhou†

* Computer Command and Control Company, 2300 Chestnut St., Su. 230, Philadelphia, PA 19103. sokolsky@cccc.com

† AlliedSignal Advanced Systems Technology Group, 9140 Old Annapolis Rd., Columbia, MD 21045. {younis,zhou}@batc.allied.com

‡ Department of Computer and Information Systems, University of Pennsylvania, Philadelphia, PA 19 104. {lee,heekwak}@saul.cis.upenn.edu

Abstract

In the recent years, formal methods has been widely recognized as effective techniques to uncover design errors that could be missed by a conventional software engineering process. This paper describes our experience with using formal methods in analyzing the Redundancy Management System (RMS) for a Space Launch Vehicle. RMS is developed by AlliedSignal Inc. for the avionics of NASA's new space shuttle, called VentureStar, that meets the expectations for space missions in the 21st century. A process-algebraicformalism is used to construct a formal specification based on the actual RMS design specifications. Analysis is performed using PARAGON, a toolsetforformal specification and verification of distributed real-time systems. A number of real-time and fault-tolerance properties were verified, allowing for some errors in the RMS pseudocode to be detected. The paper discusses the translation of the RMS specification into process algebra formal notation and results of the formal verification.

1. Introduction

In July 1996, the National Aeronautics and Space Administration (NASA) launched a very ambitious project to build the next generation of space shuttles for the 21th century. NASA wants the new spacecraft, which is named Venture&r, to be reusable for multiple missions and to be able to reach the target orbit in a single stage. One of the driving principles of the program is to reduce the cost of future space flights to encourage private companies to install their payload. After soliciting designs from different airframe companies, NASA appointed Lockheed Martin Corp. for building a proof-of-concept prototype for the VentureStar (Figure 1), called the X-33, that will eventually lead to the Reusable Launch Vehicle (LRV). AlliedSignal Aerospace is a major subcontractor to Lockheed Martin Corp. on the project to develop the avionics of VentureStar.

A quad-redundant open system architecture is to be used for the avionics of VentureStar (only triple-redundant architecture is used for the X-33 prototype). The architecture deviates radically from traditional designs by integrating multiple flight critical control within the same cage. The integrated platform hosts the flight manager and the mission manager, both are regarded as highly critical control functions on the spacecraft since they manipulate control surfaces to compensate for aerodynamic instability. To avoid losing multiple highly critical controls by a single failure, redundant components are used. Four cages with similar configuration are included to provide fault-tolerance. The cages are deployed with a redundancy management system (RMS), developed by AlliedSignal Inc. RMS, as illustrated next, provides fault detection, containment and recovery and maintains consistency between the redundant components.

Fault tolerance is critical to the operation of VentureStar. To gain additional confidence in correctness of the RMS, we undertook a formal analysis of the RMS design. Formal methods rely on mathematical semantics of the formalism to provide rigorous analysis of specifications. Numerous case studies show that formal analysis can uncover design errors that are missed by a conventional software engineering process [8]. Exhaustive verification of real-time systems is a very resource-consuming task. Given the current state-of-the-art in the area of formal methods, only systems of moderate size can be analyzed. For a formal verification project to succeed, a relatively small safety-critical component of the system has to be identified. The RMS of VentureStar provides an excellent example of such safety-critical component. A specification of the RMS, based on the pseudocode used in the design process, was constructed. The specification uses the formalism of real-time

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

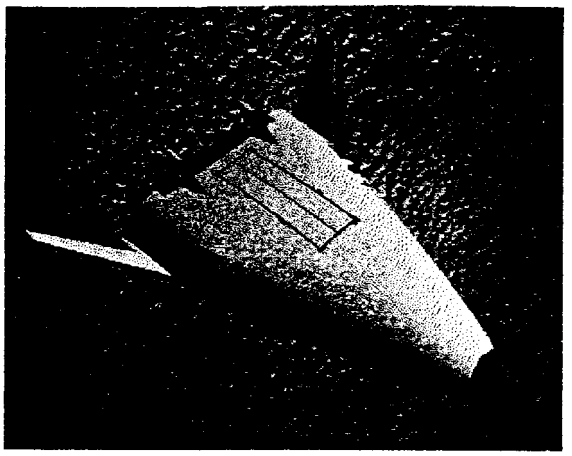

Figure 1. The Future VentureStar Reusable Launch Vehicle

process-algebra ACSR [12]. After construction, the specification was analyzed for compliance with the set of RMS requirements, which were also given a formal representation. Analysis was performed using PARAGON toolset [I] that follows the ACSR specification paradigm. The overall scheme of the approach is demonstrated in Figure 2.

Related work. Several other tools for formal analysis of system specifications are available. Among the most widely used are SPIN [9], SCR* [6] and the Concurrency Workbench [3]. Tools like HyTech [7], COSPAN [5], and SMV [13] are popular for analysis of hardware and hybrid systems. Compared to these tools, PARAGON is more oriented towards specification of real-time systems. In addition to capabilities for quantitative timing analysis, PARAGON allows notions of priorities and shared resources, common in design of real-time systems, to be used in system specifications.

The outline of the paper is as follows: Section 2 describes the RMS design and implementation, as well as requirements for the RMS. Section 3 presents PARAGON [1], the specification and verification toolset for distributed real-time systems that was used to analyze the RMS. The formal specification of the RMS and its requirements is discussed in Section 4. The paper concludes with the summary of verification results in Section 5.

2. The Fault Tolerant Architecture

Tolerance of faults, typically, can be realized in four steps [11]. The first step is to detect an error. Second the fault that caused the error has to be contained to prevent fault propagation to other system components. Then, the required diagnosis is performed to find the location (zone) of the fault. Finally, the appropriate recovery procedure is invoked, including reconfiguration if necessary. Fault-tolerance is achieved by using redundancy. Such redundancy can be a replica used in case of failure to supply the same function. A technique known as passive replication is used to mask faults by removing their effects. Faults are masked by executing voting algorithms that select the most reliable response from the replicated computers [10]. Redundancy management is necessary to synchronize the execution of multiple computers into a common clock and to vote on data to detect and mask faults. However, managing the redundancy requires overhead to keep consistency between replicas and this overhead can increase the complexity of the application development process.

The AlliedSignal research team has developed the Multicomputer Architecture for Fault Tolerance (MAFT) to support the development of real-time mission critical applications [10, 15]. The philosophy used in the MAFT architecture is to separate redundancy management and fault-tolerance support from the applications (e.g., control functions, etc.) so that the overall development complexity and effort of dependable systems can be reduced. The architecture is scalable to support as many redundant components as needed by the fault coverage requirements. Using this approach, a system developer can concentrate on system application design and can rely on the redundancy management system (RMS) to provide system executive functions such as cross-channel synchronization and data voting to achieve fault tolerance and redundancy management at the system level. This divide-and-conquer strategy is important for a complex system-engineering task so that it can be broken down into smaller and easily manageable tasks. It avoids the ad-hoc design processes for implementing fault-tolerant systems, and offers effective means for integration of design dependability into real-time, mission-critical sys-

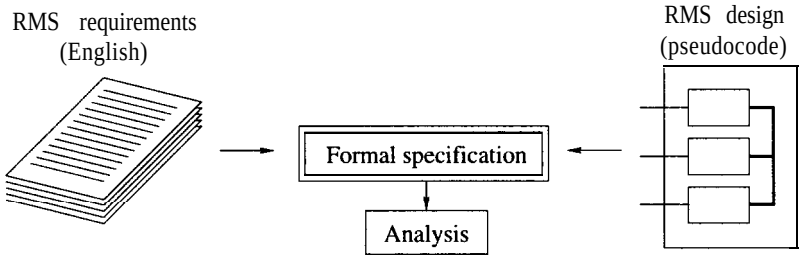

Figure 2. Formalization of the RMS

tern development.

A MAFT-based fault tolerant architecture consists of multiple processing nodes, called application processors or simply AP. Each AP performs exactly the same functions. Every node is connected to an RMS processor. All of these RMS nodes are mutually connected through direct communication links. The RMS and AP partitioning can be either logical or physical. The RMS may be a software kernel that shares the same processor with application tasks, resulting in only logical partition. The RMS may also be a hardware device that is physically separated from the AP processor. The number of redundant AP nodes is selected based on the criticality level and the types of faults that the system should handle. A sample four-channel RMS system model is depicted in Figure 3. Every channel is considered as a fault containment zone. Faulty channels will be excluded from the voting process. Thus, a fault in a channel cannot propagate to affect other healthy channels.

Using this architecture, every application function will be executed multiple times simultaneously on different nodes (four in this example). Every application function will periodically send data to the associated RMS module via the direct communication links. Every RMS module will then send that data to all other RMS nodes through dedicated communication links, called Cross Channel Data Link (CCDL). After receiving all copied data, every RMS module will perform voting and send back the voted data values that will be used by the application for further computation. The voted data can be used to mask the error generated by a faulty application node to restore system health and integrity. In addition, RMS maintains a global system health status identifying both healthy and faulty nodes based on the deviation from the voted data. Moreover, RMS maintains synchronized execution of all the redundant application processors by sending periodic synchronization messages to overcome any clock skew effect. Thus, RMS masks faults by excluding erroneous data and provides fault detection, containment, diagnosis and recovery. The RMS functions are transparent to the application processors and are available to the system developer as system services.

By providing such system service functions for the X-33 vehicle management computer, RMS plays an essential role in maintaining the availability and safety of the vehicle. Consequently, rigorous engineering design and implementation processes and fault avoidance techniques are extremely critical to verify the correctness of RMS. A single generic fault in the design or implementation of RMS may bring the whole system down regardless the number of redundant components. In addition, RMS implementation for the X-33 is mostly in software that increases the probability of subtle faults. Thus, verification and fault avoidance techniques, including the use of Formal Methods, are necessary to prove the behavior of RMS before deployment. The next section summarizes both functional and operational requirements of RMS for the X-33 VentureStar.

2.1. RMS Requirements for VentureStar

For the VentureStar, RMS has to be designed and implemented subject to a set of functional and operational requirements. Operational requirements address the behavior of RMS in both absence and presence of faults. They include performance, fault latency, errors reporting and application processor interface. On the other hand, functional requirements include the capabilities that RMS is expected to provide and the assumptions that both RMS and the applications should make about each other. The following are informal samples of the requirements:

1. RMS should complete its functional computation in a minor frame of 10 ms. A minor frame is the period of the most frequently activated task.

2. For a three or more node system, RMS should operate normally with the failure of one node.

3. The system should be able to tolerate any single fault with the following timing characteristics: (1) transient, (2) permanent, and (3) intermittent. Any of these faults should be contained in its originated node and should not be propagated to other nodes.

4. RMS should complete system recovery by excluding the faulty node in one major frame. A major frame is the period of the least common multiple of tasks' frequencies.

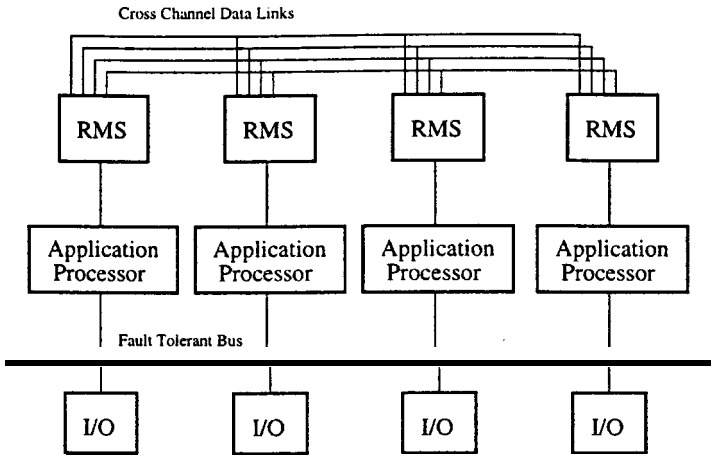

Figure 3. A four-channel RMS based fault-tolerant system

5. The system should be able to readmit a fault-free node into the operating set within one major frame in order to preserve system resources.

6. At startup RMS should synchronize with all other nodes to form the potential operating set (OPS) incrementally. All nodes in the OPS should maintain a synchronization skew of less than 0.1 ms.

7. RMS should use different voting algorithms [10, 15] for different types of data: (A) Majority voting for finite discrete data. (B) Mid-Value Selection voting for integer or floating-point numbers. (C) Mean of Medial Extremes voting for system synchronization.

8. RMS should collect application data at the minor frame boundary, vote the data, and signal the availability of the voted data with the application data ready signal before the next minor frame boundary.

9. CCDL communications should be by serial link that runs at a minimum speed of 8Mbps.

10. The CCDL shall be able to receive messages from multiple nodes (including itself) simultaneously.

11. Messages sent through the CCDL should include error detection code to detect transmission errors.

2.2. RMS Design and Implementation

Since RMS and application partitioning can be either logical or physical, both software and hardware implementation of RMS are feasible. RMS may be a software kernel that shares the same processor with application tasks, resulting in only logical partition. RMS may also be a hardware device that is physically separated from the application processor. For the X-33, a dedicated VME-based computer board separate from the application hosts RMS. A total hardware implementation of RMS makes the design less portable in spite of providing superior performance. On the other hand, a full software implementation can be easily ported to a different platform although it may not meet the timing constraints. To meet the performance goals for the VMC on the X-33 VentureStar, a hybrid approach is used by providing most of the RMS functions in software, while implementing cross channel communication between RMS nodes in hardware.

Thus, RMS consists of two parts as shown in Figure 4: (a) the Fault-Tolerant Executive (FTE) and (b) the Cross-Channel Data Link (CCDL). The FTE performs the redundancy management functions in software, whereas the CCDL performs cross-channel data communication in hardware. The FTE provides major RMS functions which include: maintaining system synchronization (Synchronizer); voting on application data and RMS internal state (Voter); error detection and fault isolation and recovery (Fault Tolerator); managing the cross channel data link (Manage CCDL); performing built-in-test at startup (Diagnostics); managing the application interface (Task Communicator); and, coordination of correct and timely operations of all the functions above (Kernel). The Cross Channel Data Link (CCDL) is designed as a mezzanine board that is seated on the VME card running the FTE. The CCDL card provides the physical interface between the redundant nodes and performs error checking on message transmission. Pseudo code is prepared for various components of the FTE and reviewed by peers. In addition, a detailed design of the CCDL including schematics is developed and verified.

RMS development follows various well-established software engineering process for software development, testing, and validation. Peer reviews are conducted during preliminary and detailed design. In addition, code inspection is per-

Part III

Probabilistic Program Correctness

Funda Ergün, Sampath ***Kannan, S. Ravi*** *Kumar,* ***Ronitt Rubinfeld, and*** *Mahesh Viswanathan:* "Spot-Checkers", in the Proceedings of the 30th ACM Symposium on the Theory of Computing — STOC 98, pages 259-268, Dallas, TX, May, 1998.

Full paper: `http://theory.stanford.edu/muri/reports/97-98/funda2.ps`

Funda Ergün, ***S. Ravi*** *Kumar,* ***and Ronitt Rubinfeld:*** "Approximate Checking of Polynomials and Functional Equations", submitted for publication. A preliminary version appeared in the Proceedings of the 37th IEEE Symposium on the Foundations of Computer Science — FOCS 98, pages 592-601, Burlington, VT, October 1996.

Full paper: `http://theory.stanford.edu/muri/reports/97-98/funda1.ps`

Spot-Checkers*

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

Abstract

On Labor Day Weekend, the highway patrol sets up spot-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 0(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- l-0505, MURI. The second author is also supported by NSF Grant CCR96-19910. 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-I-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, Lip91] 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 x 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 $\Delta(\cdot,\cdot)$ be a distancefunction. We say that* C *is an* ϵ-spot-checker *for f with distance function* A *if*

> *Given any input x andprogram 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(*(x, P(x)), (x, f(x))) = 0 and* FAIL *iffor all inputs* $y, \Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) > \epsilon$.
>
> *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 o 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 o' over domain D in $\tilde{O}(|D|)$ steps'. 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 o' 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* ***resting*** (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 $\Pr_{x\in\mathcal{D}}[f(x) \neq g(x)] \leq \epsilon$. $\mathcal{A}$ 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 $f \in 3$ and FAIL if there is no g $\in$ 3 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 d(n) suppresses polylogarithmic factors of n.

[2] In fact, the definition of property resting 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 muy 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 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($\langle x, y\rangle, \langle x,$ 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 u into v. The distance function that we use in defining our spot-checker is as follows: for all 2, y lists of elements, $\Delta(\langle x, P(x)\rangle$, (y, f(y))) is infinite if either $x \neq$ y or $|P(x)| \neq |f(y)|$ 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 $\forall x$, $|x|$ = $|P(x)|$. It 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(z)*** 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 l/S) 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 Aissorted,*** 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, 6) = 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 trivia1 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 ***ifforall k,*** $0 \leq k \leq$ lgi, $|\Gamma^-_{(i-2^k,i)}(i)| \geq \eta 2^k$ ***andforall 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 trivia1 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 $\eta 2^m - (\{(j - 2^m) - i\}) = \eta 2^m - 1$. 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 $(\eta 2^m - 1) + (\eta 2^m - l) \geq |I| = 2^m - 1 + 1$. 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 $1 - 6$. 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 Chemoff 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 Chemoff's bound. Thus the probability of (ii) is at most b/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(lgn) time will either fail a completely sorted sequence or pass a sequence that contains no increasing subsequence of length R(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, $(x_1, x_2, \ldots, x_k)$ 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 $lst_{k-1}(x_1, x_2, \ldots, x_{k-1})$. 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 $lst_1(x_1)$ to be the set of sequences in $\mathbb{Z}^{x_1}$ which are increasing.

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

$$\overbrace{\underbrace{789}_{\in lst_1(3)}\ 456123}^{\in lst_2(3,3)}\ 161718131415101112$$

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 $a(n)$ is an unbounded, increasing function of n. Without loss of generality, the checker generates $O(f(n))$ index pairs $(a_1, b_1), \ldots (a_k, b_k)$, 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

Approximate Checking of Polynomials and Functional Equations *

Funda Ergün S Ravi Kumar Ronitt Rubinfeld

Department of Computer Science

Cornell University

Ithaca, NY 14853.

July 16, 1998

Abstract

In this paper, we show how to check programs that compute polynomials and functions defined by addition theorems — in the realistic setting where the output of the program is approximate instead of exact. We present results showing how to perform approximate checking, self-testing, and self-correcting of polynomials, settling in the affirmative a question raised by [GLR+91, RS92, RS96]. We obtain this by first building approximate self-testers for linear and multilinear functions. We then show how to perform approximate checking, self-testing, and self-correcting for those functions that satisfy addition theorems, settling a question raised by [Rub94]. In both cases, we show that the properties used to test programs for these functions are both

*This work is partially supported by NSF Career grant CCR-9624552, the Alfred P. Sloan Research Award, and ONR grant N00014-97-1-0505. The first and second authors are also supported by NSF grant DMI-91157199. The third author is also supported by the grant No. 92-00226 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel. Part of this research was conducted while visiting the M.I.T. Lab. for Computer Science. A preliminary version of this paper appeared in the 37th IEEE Conference on Foundations of Computer Science, 1996.

robust (in the approximate sense) and stable. Finally, we explore the use of reductions between functional equations in the context of approximate self-testing. Our results have implications for the stability theory of functional equations.

1 Introduction

Program checking was introduced by Blum and Kannan [BK89] in order to allow one to use a program safely, without having to know apriori that the program is correct on all inputs. Related notions of self-testing and self-correcting were further explored in [BLR93, Lip91]. These notions have proved to be very powerful from a practical point of view (c.f., [BW94]) and from a theoretical angle (c.f., [AS92, ALM+92]) as well. The techniques used usually consist of tests performed at run-time which compare the output of the program either to a predetermined value or to a function of outputs of the same program at different inputs. In order to apply these powerful techniques to actual programs, however, several issues dealing with *precision* need to be dealt with. The standard model, which considers an output to be wrong even if it is off by a very small margin, is too strong to make practical sense, due to reasons such as: (1) in many cases, the algorithm is only intended to compute an approximation, e.g., Newton's method; (2) representational limitations and roundoff/truncation errors are inevitable in real-valued computations; (3) the representation of some fundamental constants (e.g., π = 3.14159. . .) is inherently imprecise.

The framework presented by [GLR+91, ABC+93] accommodates these inherently inevitable or acceptably small losses of information by overlooking small precision errors while detecting actual "bugs", which manifest themselves with greater magnitude. Given a function f, a program P that purports to compute f, and an error bound Δ, if $|P(x) - f(x)| \leq \Delta$ (denoted $P(x) \approx_\Delta f(x)$) under some appropriate notion of norm, *we* say *P(x)* is *approximately correct*. *Approximate result checkers* test if *P* is approximately correct for a given input *x*. *Approximate self-testers* are programs that test if *P* is approximately correct for most inputs. *Approximate self-correctors* take programs that are approximately correct on most inputs and turn them into programs that are approximately correct on every input.

DOMAINS. We operate on finite subsets of fixed point arithmetic that we refer to as *finite rational domains*. For n, s $\in \mathbb{Z}^+$, $\mathcal{D}_{n,s} \stackrel{\text{def}}{=} \{\frac{i}{s} : |i| \leq \text{n}, \text{i} \in \mathbb{Z}\}$. Usually, $s = 2^l$ where l is the precision. We allow s and *n* to vary for generality. For a domain $\mathcal{D}$, let $\mathcal{D}^+$ and $\mathcal{D}^-$ denote the positive and negative elements in $\mathcal{D}$.

TESTING USING PROPERTIES. There are many approaches to building self-testers. We

illustrate one paradigm that has been particularly useful. In this approach, in order to test if a program P computes a function f on most inputs, we test if P satisfies certain properties of f.

As an example, consider the function f(z) = 2x and the property "$f(x + 1) = f(x) + 2$" that f satisfies. One might pick random inputs x and verify that $P(x + 1) = P(x) + 2$. Clearly, if for some x, $P(x + 1) \neq P(x) + 2$, then P is incorrect. The converse, however, is not true. In particular, there exists a P (for instance, $P(x) = 2x \bmod K$ for some large K) such that: (i) with high probability, P satisfies the property at random x and hence will pass the test, and (ii) there is no function that satisfies the property for all x such that P agrees with this function on most inputs. Thus we see that this method, when used naively, does not yield a self-tester that works according to our specifications. Nevertheless, this approach has been used as a good heuristic to check the correctness of programs [Cod91, CS91, Vai93].

As an example of a property that does yield a good tester, consider the linearity property "$f(x + y) = f(x) + f(y)$", satisfied only by functions mapping $\mathcal{D}_{n,s}$ to $\mathbb{R}$ of the form f(x) = cx, $c \in \mathbb{R}$. If, by random sampling, we conclude that the program P satisfies this property for most x, y, it can be shown that P agrees with a linear function g on most inputs [BLR93, Rub94]. We call linearity, and any property that exhibits such behavior, a *robust property*.

We now describe more formally how to build a self-tester for a class 3 of functions that can be characterized by a robust property. Our two-step approach is: (i) test that P satisfies the robust property *(property testing)*, and (ii) check if P agrees with a *specific* member of 3 *(equality testing)*. The success of this approach depends on finding robust properties which are both easy to test and lead to efficient equality tests.

We first consider robust properties in more detail. Suppose we want to test the program on the domain $\mathcal{D}_{n,s}$. Then we allow calls to the program on a larger domain $\mathcal{D}_{\tau(n,s)}$, where $\tau : \mathbb{Z}^2 \to \mathbb{Z}^2$ is a fixed function that depends on the structure of $\mathcal{I}$. Ideally, we would like $\tau(n, s) = (n, s)$, i.e., $\mathcal{D}_{\tau(n,s)} = \mathcal{D}_{n,s}$. But, for technical reasons, we allow $\mathcal{D}_{\tau(n,s)}$ to be a proper, but not too much larger, superset of $\mathcal{D}_{n,s}$ (in particular, the description size of an element in $\mathcal{D}_{\tau(n,s)}$ should be polynomial in the description size of an element in $\mathcal{D}_{n,s}$).[1] A *property*

[1] Alternatively, one could test the program over the domain $\mathcal{D}_{n,s}$ and attempt to infer rhe correctness of the program on most inputs from $\mathcal{D}_{n',s'}$, where $\mathcal{D}_{n',s'}$ is a large subdomain of I?,,,,.

Part IV

Programming Languages

Iliano Cervesato: "Proof-Theoretic Foundation of Compilation in Logic Programming Languages", in the Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming — JICSLP '98, (J. Jaffar editor), pages 115–129, MIT Press, Manchester, UK, June 1998.

Full paper: http://www.stanford.edu/~iliano/papers/jicslp98.ps.gz

Stephen Freund and John C. Mitchell: "A Type System for Object Initialization in the Java Bytecode Language", in the Proceedings of the ACM Symposium on Object-oriented Programming: Systems, Languages and Applications — OOPSLA'98, Vancouver, Canada, October, 1998 (to appear).

Full paper: ftp://theory.stanford.edu/pub/jcm/papers/jvm-oopsla-98.ps

Proof-Theoretic Foundation of Compilation in Logic Programming Languages

Iliano Cervesat 0
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
iliano@cs.stanford.edu

Abstract

Commercial implementations of logic programming languages are engineered around a compiler based on Warren's Abstract Machine (WAM) or a variant of it. In spite of various correctness proofs, the logical machinery relating the proof-theoretic specification of a logic programming language and its compiled form is still poorly understood. In this paper, we propose a logic-independent definition of compilation for logic programming languages. We apply this methodology to derive the first cut of a compiler and the corresponding abstract machine for the language of hereditary Harrop formulas and then for its linear refinement.

1 Introduction

Compiled logic programs run over an order of magnitude faster than their interpreted source and constitute therefore a key step to combining the advantages of the declarative nature of logic programming with the efficiency requirements of full-scale applications. For this reason, commercial implementations of logic programming languages come equipped with a compiler to translate a source program into an intermediate language, and an abstract machine to execute this compiled code efficiently. Most systems are based on ***Warren's Abstract Machine*** (WAM) [1, 22], first developed for ***Prolog***. The WAM has now been adapted to other logic programming languages such as $CLP(\mathcal{R})$ [10] and ***PROTOS-L*** [2]. Extensions to $\lambda Prolog$ [14] are under way [12, 16, 17], but no similar effort has been undertaken for other advanced logic programming languages such as *Lolli* [9] or *Elf* [20].

Warren's work appears as a carefully engineered construction, but, for its very pioneering nature, it lacks any logical status. This contrasts strongly with the deep roots that the interpretation semantics of logic programming has in logic and proof-theory [15]. Indeed, the instruction set of the WAM hardly bears any resemblance to the connectives of the logic underlying ***Prolog*** and seems highly specialized to this language. As a result, the WAM "resembles an intricate puzzle, whose many pieces fit tightly together in a

Appeared in the ***Proceedings of the 1998 Joint International*** Conference and Symposium on Logic Programming — *JICSLP'98* (J. Jaffar editor), pp ??-??, MIT Press, Manchester, UK, 16–19 June 1998.

miraculous way" [3], understanding it is complex in spite of the availability of excellent presentations [1], and designing WAM-like systems for other languages is a major engineering project. Several authors have proved the correctness of the WAM with respect to specifications of ***Prolog***'s interpretation semantic [3, 21]. However, these results shed very little light on the logical reading of the WAM since they start from highly procedural presentations of the semantic of Horn clauses, as SLD-resolution for example. Adaptations of these proofs to the aforementioned extensions of the WAM to other logic programming languages [2, 4] suffer from the same problem. Therefore, differently from the case of functional programming for example [13,18], there is no logical account of compilation for logic programs.

In this paper, we give a general definition of compilation that parallels and extends the notion of abstract logic programming language [15]. More specifically, we propose a proof-theoretic characterization of the compilation process based on the duality between left and right rules in a sequent calculus presentation of a logic. In doing so, we do not commit to any particular logic but consider any formalism whose proof theory obeys a minimal set of properties [15]. We use the logic of hereditary Harrop formulas [15] and its linear variant [9] as examples. The high-level rules of an abstract logic programming language leave several implementation choices open, such as the order in which conjunctive goals should be solved and how to instantiate variables. The same will happen in the compiled language. However, similarly to the case of abstract logic programming languages, our abstract compilation scheme can be refined to adopt specific strategies (e.g. left-to-right subgoal selection rule and unification). Modularity is achieved in this way.

The main contributions of this paper are: (1) the individuation of the logic underlying the WAM and a logical justification of the compilation process, (2) the definition of an abstract, logic-independent and modular notion of compilation for logic programming languages, and possibly (3) the first steps toward a logic-based theory of compilation for logic programming languages. We also hope that our approach will help make compilation an integral part of the design of new logic programming languages rather than the collateral engineering task it is now.

This paper is organized as follows. In Section 2, we recall the definition of abstract logic programming language and introduce the logic of hereditary Harrop formulas as an example. We describe our abstract notion of compilation in Section 3, apply it to our case study and prove correctness results. In Section 4, we further exemplify our approach on a logic programming language based on linear logic. Section 5 outlines directions of future work.

2 Abstract Logic Programming Languages

Computation in logic programming is achieved through proof search. Given a goal A to be proved in a ***program*** Γ, we want to be able to interpret the connectives of A as ***search*** *directives* and the clauses in Γ as specifications

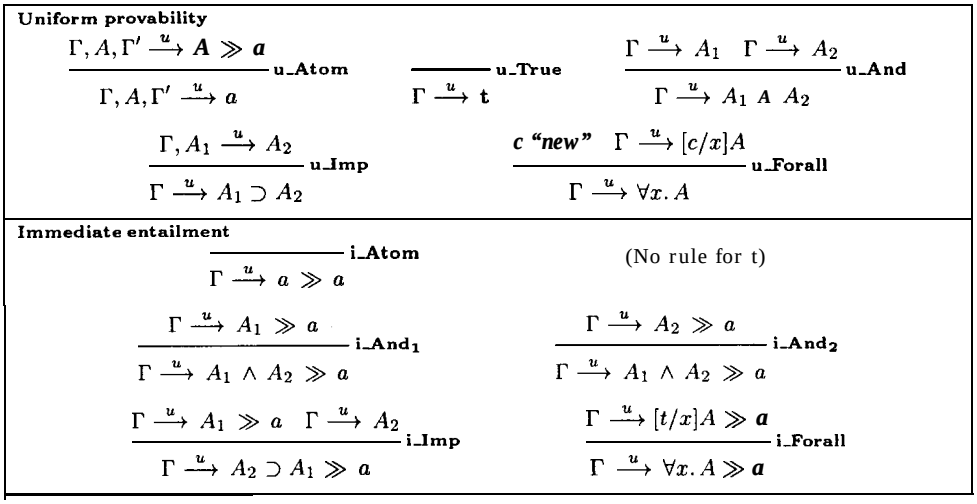

Figure 1: Uniform Deduction System for $\mathcal{L}_{HH}$.

of how to continue the search when the goal is atomic. These desiderata are given without mentioning the logic the program and the goal belong to.

A proof in any logic is goal-oriented if every compound goal is immediately decomposed and the program is accessed only after the goal has been reduced to an atomic formula. A proof is ***focused*** if every time a program formula is considered, it is processed up to the atoms it defines without need to access any other program formula. A proof having both these properties is *uniform*, and a formalism such that every provable goal has a uniform proof is called an ***abstract*** logic programming *language*, abbreviated ***ALPL*** [15]. An ALPL is conveniently described as a pair $(\mathcal{L}, \longrightarrow)$ consisting of a language $\mathcal{L}$ and an associated notion of (uniform) derivability $\longrightarrow$.

The language of ***hereditary*** *Harrop* formulas, abbreviated $\mathcal{L}_{HH}$, is the largest freely generated fragment of intuitionistic logic that passes the above ALPL test [15]. It is the logic underlying the programming language $\lambda Prolog$ [14] and embeds the language of Horn clauses on which ***Prolog*** is based. Formulas and programs are defined by means of the following grammar:

Formulas: $A ::= a \mid \mathbf{t} \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid \forall x.\, A$
Programs: $\Gamma ::= . \mid \Gamma, A$

where a ranges over atomic formulas. We shall treat programs as sequences and omit the leading "$\cdot$" whenever a program is not empty. Moreover, we write t to indicate terms and denote the capture avoiding substitution of a term t for x in a formula A as $[t/x]A$. Notice that we do not specify the underlying language of terms. This is irrelevant to our development: it can be first-order, higher-order as in $\lambda Prolog$, or be based on any equational theory (which would yield corresponding *constraint* ***logic programming languages).*** Propositional $\mathcal{L}_{HH}$ can be handled as well once we omit the quantifiers $\forall x.\, A$. However, we do not admit quantification over atomic formulas.

Syntactically richer definitions of the logic of hereditary Harrop formulas exist [14,15]. They are obtained by allowing the language of programs to be different from the language of goals. The former differs in minor points from the grammar displayed above. The latter is extended with numerous other logical operators. It is not a coincidence that these additional constructs correspond to the connectives and quantifiers into which program formulas will be compiled in the next section. This syntactic extension does not however alter the expressiveness of this logic.

Figure 1 displays the traditional sequent calculus rules for $\mathcal{L}_{HH}$ written in such a way that every proof is uniform. The specification of the ***uniform provability*** judgment, written $\Gamma \xrightarrow{u} A$, includes all the right sequent rules for this logic and calls the ***immediate entailment*** judgment $\Gamma \xrightarrow{u} A \gg a$ when the goal is atomic. Immediate entailment isolates a clause and decomposes it as prescribed by the left sequent rules of $\mathcal{L}_{HH}$ until an axiom applies. The cut-elimination theorem and simple permutations of inference rules suffice to show that any sequent derivation can be transformed into a uniform proof that uses only these rules [15]. Therefore $\mathcal{L}_{HH}$ is an ALPL.

The rules in Figure 1 present three main forms of non-determinism. First, the order in which the two premises of rules **u_And** and **i_Imp** should be proved is left unspecified ***(conjunctive*** non-determinism). Second. no criterion is given to select the program formula in rule **i_Atom** nor to choose among rules $\mathbf{i\_And_1}$ and $\mathbf{i\_And_2}$ (disjunctive ***non-determinism).*** Third, no strategy is specified to construct the term t in rule **i_Forall** (*existential non-determinism).* A concrete implementation must resolve these points.

3 Abstract Compilation

The search for a uniform proof consists of the alternation of two distinct phases: goal decomposition, where right sequent rules are applied according to the structure of the goal, and, once the goal is atomic, clause ***decomposition*** where left sequent rules are systematically used on the selected formula in the program. These two operational modes are clearly distinguished in $\mathcal{L}_{HH}$ by means of the two forms of judgments that describe its semantics (Figure 1). From a logic programming perspective, the connectives appearing in the goal are search directives and therefore goal decomposition is where the bulk of the action takes place. Clause decomposition can instead be viewed as a preparatory phase.

The objective of compilation is to separate these two phases so that all the preparatory steps are performed before any search begins [11]. An ***abstract logic programming compilation system, ALPCS*** for short, consists therefore of ***a source ALPL*** $(\mathcal{L}, \longrightarrow)$, ***an intermediate language*** $\mathcal{L}^c$ which is itself an ALPL with the characteristic that its notion of derivability $\xrightarrow{c}$ consists of right sequent rules only, and a ***compilation function*** $\gg$ that maps programs and goals in $\mathcal{L}$ to compiled ***programs*** and ***compiled goals*** in $\mathcal{L}^c$, respectively.

The intermediate XLPL $(\mathcal{L}^c, \xrightarrow{c})$ and the compilation function can be

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

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 interpreted 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 interpreted. As illustrated by previous attacks on the Java Virtual Machine, these tests, which include type correctness, 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 Java bytecode and this bytecode 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 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. 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 the bytecode 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. 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. The need for conservative analysis stems from the undecidability of the halting problem, as well as efficiency considerations. Specifically, since most bytecode is 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

***Supported in part by NSF grants CCR-9303099 and CCR-9629754, ONR MURI Award N00014-97-1-0505, and a NSF Graduate Research Fellowship.**

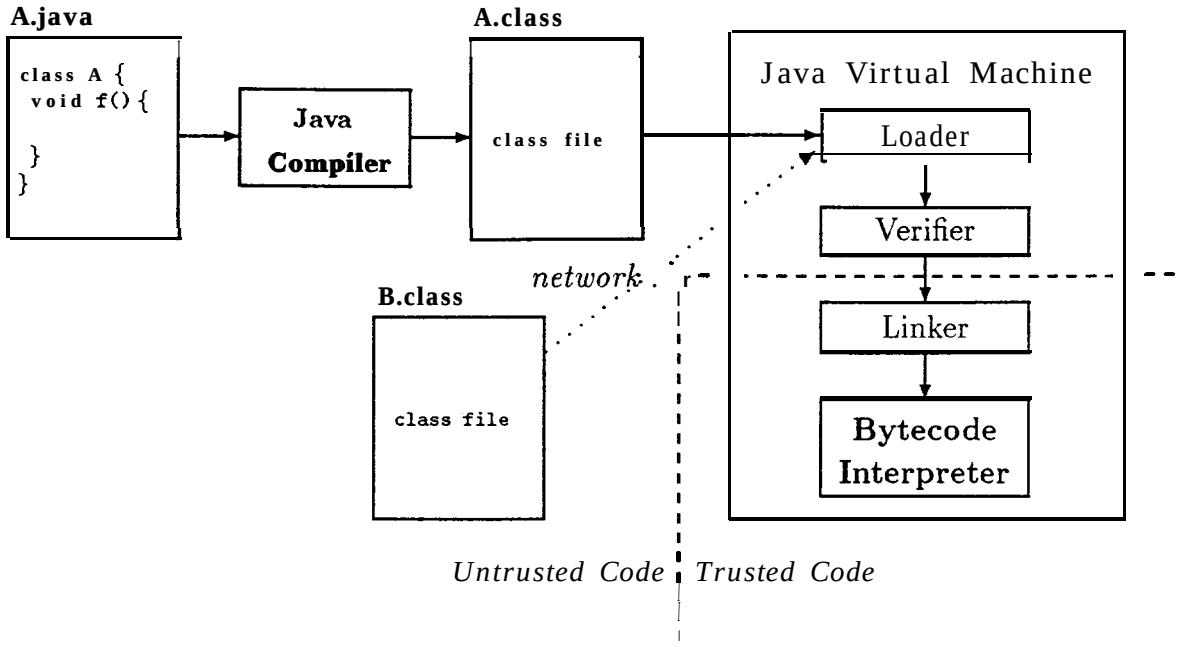

Figure 1: The Java Virtual Machine

`try-finally` construct. While some amount of type information is included in JVML to make type-checking possible, there are some high-level properties of Java source code that are not easy to detect in the resulting bytecode. 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 in bytecode programs [SA98].

Another example is the initialization of objects before use. While it is clear from the Java source language statement

$$A\ x = new\ A(\langle parameters \rangle)$$

that the A class constructor will be called before any methods can be invoked through the pointer 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 the bytecode, these 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 of 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 early forms of the Java Virtual Machine illustrate the importance of the bytecode verifier for system security. To cite one specific example, a hug 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 [DFW96]. 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 [SA98], 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 gener-

ally resisted the temptation to do so since we hoped to gain some understanding of the strength and limitations of existing verifier implementations.

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 commands 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 repairs 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_i, 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 JVML_i, 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 some environment-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. This is 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. A convention of parameter passing for the stack-based architecture is that 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 following bytecode sequence:

```
1:  new #1 <Class Point>
2:  nea #1 <Class Point>
3:  dup
4:  iconst_3
5:  invokespecial #4 <Method Point(int)>
6:  invokevirtual #5 <Method void Print()>
```

When line 5 is reached during execution, there will be two different uninitialized Point objects. If the bytecode verifier is to check object initialization statically, it must be able to determine which references point to the object that is initialized at line 5 and which point to the remaining uninitialized object. Otherwise, the verifier would either prevent use of an initialized object or allow use of an uninitialized one. (The bytecode program above is valid and accepted by verifiers using the static analysis described below.)

Sun's Java Virtual Machine Specification [LY96] describes the alias analysis used by the Sun JDK verifier. For each line of the bytecode program, some status information is recorded for every local variable and stack location. When a location points to an object that is known not to be initialized in all executions reaching this statement, the status will include not only the prop erty ***uninitialized,*** but also the line number on which the uninitialized object would have been created. As references are duplicated on the stack and stored and loaded in the local variables, the analysis also duplicates these line numbers, and all references having the same line number are assumed to refer to the same object.

When an object is initialized, all pointers that refer to objects created at the same line number are set to ***initialized.*** In other words, ail references to uninitialized objects of a certain type are partitioned into equivalence classes according to what is statically known about each reference, and all references that point to uninitialized objects created on the same line are assumed to be aliases. This is a very simple and highly conservative form of aliasing analysis; far more sophisticated methods might be considered. However, the approach can be implemented efficiently and it is sufficiently accurate to accept bytecode produced by standard compilers.

Our specification of statically-correct Java bytecode in Section 4 uses the same form of aliasing analysis as the Sun JDK verifier. Since our approach is type based, the status information associated with each reference is recorded as part of its type.

One limitation of aliasing analysis based on line numbers is that no verifiable program can ever be able to reference two objects allocated on the same line, without first initializing at least one of them. If this situation were to occur, references would exist to two different objects from the same static aliasing-equivalence class. Unfortunately, there was an oversight in this regard in the development of the Sun verifier. which allowed such a case to exist (as of version 1.1.4). As discussed in Section 7, aliasing based on line numbers makes it problematic for a subroutine to return an uninitialized object.

3 JVML;

This section describes the JVML_i language, a subset of JVML encompassing basic constructs and object initialization. Although this language is much smaller than JVML, it is sufficient to study object initialization and formulate a sound type system encompassing the static analysis described above. The run-time environment for JVML_i consists only of an operand stack and a finite set of local variables. A JVML_i program will be a sequence of instructions drawn from the following list:

instruction ::= push 0 | inc | pop
| if ***L***
| store ***x*** | load ***x***
| new σ | **init** σ | use σ
| halt

where ***x*** is a local variable name, σ is an object type, and ***L*** is an address of another instruction in the program. Informally, these instructions have the following effects:

push 0: pushes integer 0 onto the stack.

inc: adds one to the value on the top of the stack, if that value is an integer.

pop: removes the top element from the stack, provided that the stack is not empty.

if ***L:*** if the top element on the stack is not 0, execution jumps to instruction ***L.*** Otherwise, execution steps to the next sequential instruction. This assumes that the top element is an integer.

store ***x*:** removes a value from the top of the stack and stores it into local variable ***x***.

load ***x*:** loads the value from local variable ***x*** and places it on the top of the stack.

halt: terminates program execution.

new σ**:** allocates a new, uninitialized object of type σ and places it on the stack.

init σ**:** initializes the object on top of the operand stack, which must be a previously uninitialized object obtained by a call to new σ. This represents calling the constructor of an object. In this model, we assume that constructors always properly initialize their argument and return. However, as described in Section 6. there are several additional properties which must be checked to verify that constructors do in fact behave correctly.

Part V

Temporal Reasoning

Iliano Cervesato, *Massimo* ***Franceschet, and Angelo Montanari:*** "Event Calculus with Explicit Quantifiers", in the Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning — TIME'98, (R. Morris, L. Khatib editors), pages 81-88, IEEE Computer Society Press, Sanibel Island, FL, May 1998.

Full paper: `http://www.stanford.edu/~iliano/papers/time98.ps.gz`

Iliano *Cervesato,* **Massimo** ***Franceschet, and Angelo Montanari:*** "The Complexity of Model Checking in Modal Event Calculi with Quantifiers", in the Proceedings of the Sixth International conference on Principles of Knowledge Representation and Reasoning — KR'98, (A. G. Cohn, L. K. Shubert, and S. C. Shapiro, editors), pages 368–379, Morgan Kaufmann publishers, Trento, Italy, June 1998.

Full paper: `http: //www.stanford.edu/~iliano/papers/kr98.ps.gz`

Event Calculus with Explicit Quantifiers*

Iliano Cervesato† , Massimo Franceschet‡ , and Angelo Montanari‡

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

‡ Dipartimento di Matematica e Informatica
Universit& di Udine
Via delle Scienze, 206 – 33100 Udine, Italy
franzesc@dimi.uniud.it; montana@dimi.uniud.it

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. We extend the range*** *of* ***queries accepted by EC, so*** *far* ***limited to boolean combinations*** *of* MVI ***verification or computation requests, to*** admit ***arbitrary quantification over events and properties. We demonstrate the added expressive power by encoding*** a ***medical diagnosis problem as a case study.*** Moreover, we give ***an implementation*** *of* ***this formalism and analyze the*** computational ***complexity*** *of* ***the extended*** calculus.

1 Introduction

The ***Event Calculus,*** abbreviated ***EC*** [5], 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.

*The first author was supported by ONR grant N00014-97-1-0505, Multidisciplinary University Research Initiative Semantic ***Consistency in Information Exchange.*** The work of the third author was partially supported by the CNR project ***Programmazione logica: strumenti per analisi e trasformazione di programmi; tecniche di ingegneria del software; estensioni con vincoli, concorrenza ed oggetti (STE).***

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 [2]. This has had the effects of establishing a semantic reference against which to verify the correctness of implementations [2], of casting ***EC*** as a model checking problem [3], and of setting the ground for studying the complexity of this problem, which was proved polynomial [1]. Extensions of this model have been designed to accommodate constructs intended to enhance the expressiveness of ***EC.*** In particular, modal versions of ***EC*** [2], the interaction between modalities and connectives [3], and preconditions [4] have all been investigated in this context.

In this paper, we continue the endeavor to enhance the expressive power of ***EC*** by considering the possibility of quantifying over events and properties in queries. We also admit boolean connectives and requests to verify the relative order of two events. We show that the resulting language, that we call ***QCEC,*** can effectively be used to encode interesting problems in medical diagnosis. Moreover, we provide an elegant implementation in the higher-order logic programming language $\lambda Prolog$ [6] and prove its soundness and completeness. Finally, we analyze the complexity of the model checking problem involving this language.

The main contributions of this work are: (1) the extension of the event calculus with quantifiers; (2) permitting queries to mention ordering information; and (3) the use of the higher-order features of modern logic programming languages in temporal reasoning.

This paper is organized as follows. In Section 2, we formalize ***QCEC.*** Section 3 is devoted to exemplifying how this calculus can adequately model certain medical diagnosis problems. In Section 4, we briefly introduce the logic programming language $\lambda Prolog$, give an implementation of ***QCEC*** in it and prove the sound-

Appeared in the ***Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning — TIME'98*** (R. Morris, L. Khatib editors), pp. 81-88, IEEE Computer Society Press, Sanibel Island, FL, 16-17 May 1998.

ness and completeness of the resulting program. In Section 5, we analyze the complexity of ***QCEC.*** We outline directions of future work in Section 6.

2 Event Calculus with Quantifiers

In this section, we first recall the syntax and semantics of the Event Calculus, ***EC*** for short (2.1). We then extend this basic definition to give a semantic foundation to the Event Calculus with Connectives and Quantifiers, abbreviated ***QCEC (2.2).***

2.1 EC

The Event Calculus ***(EC)*** [5] and the extension 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*** [2], defined as follows:

Definition 2.1 ***(EC-structure)***

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

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

The temporal aspect of ***EC*** is given by the order in which events happen. For the sake of generality [2], we admit scenarios in which the occurrence times of events are unknown or in which the relative order of event happenings is incomplete. Clearly our argument specializes to the common situation where every event has an associated occurrence time. We however require the temporal information to be consistent so that an event cannot both precede and follow some other event. In its most basic form, ***EC*** does not take the evolution of the event ordering into account, but operates on temporal snapshots. We can then formalize the time-dependent aspect of a scenario modeled by ***EC*** by means of a (strict) ***partial order*** for the involved event occurrences. We write $W_{\mathcal{H}}$ for the set of all partial orders over the set of events $\mathbf{E}$ in 'l-l, use the letter w to denote individual orderings and write $e_1 <_w e_2$ to indicate that e_1 precedes e_2 in the ordering w. For reasons of efficiency, implementations usually represent the temporal information of an ***EC*** problem as ***a binary acyclic relation o*** from which w can be recovered by taking its transitive closure, written o^+.

Given a structure $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ and an event ordering w, we call the pair $(\mathcal{H}, \mathrm{w})$ an ***EC-problem. EC*** permits inferring the ***maximal validity intervals,*** or ***MVIs***, over which a property ***p*** holds uninterruptedly. We represent an MVI for ***p*** as $p(e_i, e_t)$, where e_i and e_t are the events that respectively initiate and terminate the interval over which ***p*** holds maximally. Consequently, we adopt ***as*** the ***query language*** of an ***EC*** problem $(\mathcal{H}, \mathrm{w})$ the set

$$\mathcal{L}_{\mathcal{H}}(EC) = \{p(e_1, e_2) : p \in \mathrm{P} \text{ and } e_1, e_2 \in \mathbf{E}\}$$

of all such property-labeled intervals over $\mathcal{H}$. We interpret the elements of $\mathcal{L}_{\mathcal{H}}(EC)$ as propositional letters and the task performed by ***EC*** reduces to deciding which of these formulas are MVIs in w and which are not. This is a model checking problem.

In order for $p(e_1, e_2)$ to be an MVI relative to the event ordering w, it must be the case that $e_1 <_w e_2$. Moreover, e_1 and e_2 must witness the validity of the property ***p*** at the ends of this interval by initiating and terminating ***p,*** respectively. These requirements are enforced by conditions (i), (ii) and (iii), respectively, in the definition of valuation given below. The maximality requirement is caught by the negation of the meta-predicate $br(p, e_1, e_2, \mathrm{w})$ in condition ***(iv),*** which expresses the fact that the truth of an MVI must not be ***broken*** by any interrupting event. Any event e which is known to have happened between e_1 and e_2 in w and that initiates or terminates a property that is either ***p*** itself or a property exclusive with p interrupts the validity of $p(e_1, e_2)$. These observations are formalized as follows.

Definition 2.2 ***(Intended model of EC)***

Let $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ ***be a EC-structure and*** $w \in W_{\mathcal{H}}$. ***The*** intended EC-model of $(\mathcal{H}, \mathrm{w})$ is ***the propositional*** valuation $v_{(\mathcal{H},w)} \subseteq \mathcal{L}_{\mathcal{H}}(EC)$, ***where*** $p(e_1, e_2) \in v_{(\mathcal{H},w)}$ ***if and only if***

i. $e_1 <_w e_2$;

ii. $e_1 \in [p\rangle$;

iii. $e_2 \in \langle p]$;

iv. $br(p, e_1, e_2,$ ***w) does*** not ***hold, where*** $br(p, e_1, e_2,$ ***w) abbreviates:***

there exists an event $e \in \mathbf{E}$ ***such that*** $e_1 <_w e$, $e <_w e_2$ and ***there exists a property*** $q \in \mathrm{P}$ ***such thut*** $e \in [q\rangle$ or $e \in \langle q]$, *and zither* $]p, q[$ ***orp = q.*** □

2.2 QCEC

We will now enrich the query language of the Event Calculus with universal and existential quantifiers over both events and properties. In order to make the resulting formalism more interesting, we further add boolean connectives and the possibility of testing the relative order of events. Indeed, a logic programming implementation of ***EC*** can emulate existential quantification over individual formulas in $\mathcal{L}_{\mathcal{H}}$ ***(EC)*** by means of unification, and moreover, universally quantified formulas in this language always have trivial solutions. We call the resulting formalism the ***Event Calculus with*** Connectives ***and Quantifiers,*** or ***QCEC*** for short.

The addition of connectives, precedence testing and unrestricted quantification over events gives ***QCEC*** a considerably improved expressive power with respect to ***EC.*** This will be demonstrated in Section 3 where we will be able to encode a medical diagnosis problem that cannot be easily tackled by ***EC.*** The computational complexity of the extended calculus remains polynomial in the numbers of events, but becomes exponentials in the quantifiers nesting of the query, as we will see in Section 5. However, in realistic applications the query size is likely to be much smaller than the number of recorded events.

Quantifiers over property do not appear to enhance significantly the expressiveness of ***EC*** due to the tight relation between properties and events, hard-coded in the initiation and termination maps. However, we expect substantial benefits in a language that admits the use of preconditions [4]. We nonetheless treat property quantifiers since they are handled similarly to quantification over events.

In order to accommodate quantifiers, we need to extend the query language of an ***EC*** problem $(\mathcal{H}, \mathrm{w})$, with $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$, in several respects. We first assume the existence of infinitely many event ***variables*** that we denote ***E,*** possibly subscripted. We similarly need a countable set of ***property variables,*** indicated with the letter ***P*** variously decorated. We write $\bar{e}$ for a syntactic entity that is either an event in **E** or an event variable. We adopt a similar notation in the case of properties. The query language of ***QCEC,*** denoted $\mathcal{L}_{\mathcal{H}}(QCEC)$, is then the set of closed formulas generated by the following grammar:

$$\varphi ::= \bar{p}(\bar{e}_1, \bar{e}_2) \mid \bar{e}_1 < \bar{e}_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall E.\varphi \mid \exists E.\varphi \mid \forall P.\varphi \mid \exists P.\varphi.$$

where $\bar{e}_1 < \bar{e}_2$ denotes the test of whether $\bar{e}_1$ precedes $\bar{e}_2$. Observe that $\forall$ and $\exists$ have been overloaded to indicate quantification over both events and properties; the nature of the syntactic variable that follows these symbols allows disambiguating their use. In addition to the operators above, we also admit implication $(\supset)$ as a derived connective, where $\varphi_1 \supset \varphi_2$ is classically defined as $\neg\varphi_1 \vee \varphi_2$.

The notions of free and bound variables are defined as usual and we identify formulas that differ only by the name of their bound variables. We write $[e/E]\varphi$ for the substitution of an event $e \in \mathbf{E}$ for every free occurrence of the event variable ***E*** in the formula φ, and similarly for properties. Notice that this limited form of substitution cannot lead to variable capture.

We now extend the definition of intended model of an EC-problem $(\mathcal{H}, w)$ from formulas in $\mathcal{L}_{\mathcal{H}}(EC)$ to objects in $\mathcal{L}_{\mathcal{H}}(QCEC)$. To this aim, we need to define the notion of validity for the new constructs of ***QCEC.***

Definition 2.3 ***(Intended model of*** $QCEC$***)***

Let $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ ***be*** an ***EC-structure and w an event ordering. The*** intended QCEC-model of 7-i and w is ***the classical*** model $\mathcal{I}_{(\mathcal{H},w)}$ ***built on top of the valuation*** $\upsilon_{(\mathcal{H},w)}$. ***Given a (closed) formula*** $\varphi \in \mathcal{L}_{\mathcal{H}}(QCEC)$, ***the truth of*** φ ***at*** $\mathcal{I}_{(\mathcal{H},w)}$ ***, denoted as*** $\mathcal{I}_{(\mathcal{H},w)} \models \varphi$, ***is inductively defined as follows:***

$\mathcal{I}_{(\mathcal{H},w)} \models p(e_1, e_2)$ *iff* $p(e_1, e_2) \in \upsilon_{(\mathcal{H},w)}$;
$\mathcal{I}_{(\mathcal{H},w)} \models e_1 < e_2$ *iff* $e_1 <_w e_2$;
$\mathcal{I}_{(\mathcal{H},w)} \models \neg\varphi$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \not\models \varphi$;
$\mathcal{I}_{(\mathcal{H},w)} \models \varphi_1 \wedge \varphi_2$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \varphi_1$ *and* $\mathcal{I}_{(\mathcal{H},w)} \models \varphi_2$;
$\mathcal{I}_{(\mathcal{H},w)} \models \varphi_1 \vee \varphi_2$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \varphi_1$ *or* $\mathcal{I}_{(\mathcal{H},w)} \models \varphi_2$;
$\mathcal{I}_{(\mathcal{H},w)} \models \forall E.\varphi$ *iff for all* $e \in \mathbf{E}$, $\mathcal{I}_{(\mathcal{H},w)} \models [e/E]\varphi$;
$\mathcal{I}_{(\mathcal{H},w)} \models \exists E.\varphi$ *iff* there ***exists*** $e \in \mathbf{E}$ ***such that*** $\mathcal{I}_{(\mathcal{H},w)} \models [e/E]\varphi$;
$\mathcal{I}_{(\mathcal{H},w)} \models \forall P.\varphi$ *iff for all* $p \in \mathbf{P}$, $\mathcal{I}_{(\mathcal{H},w)} \models [p/P]\varphi$;
$\mathcal{I}_{(\mathcal{H},w)} \models \exists P.\varphi$ ***iff there exists*** $p \in \mathbf{P}$ ***such that*** $\mathcal{I}_{(\mathcal{H},w)} \models [p/P]\varphi$. □

The well-foundedness of this definition derives from the observation that if VE. φ is a closed formula, so is $[e/E]\varphi$ for every event $e \in \mathbf{E}$, and similarly for the other quantifiers.

A universal quantification over a finite domain can always be expanded as a finite sequence of conjunctions. Similarly an existentially quantified formula is equivalent to the disjunction of all its instances. The following lemma, whose simple proof we omit, applies these principles to ***QCEC.***

Lemma 2.4 ***(Unfolding*** quantifiers)

Let $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ ***be an EC-structure, with*** $\mathbf{E} = \{e_1, \ldots, e_n\}$ ***and*** $\mathbf{P} = \{p_1, \ldots, p_m\}$. ***Then, for every*** $w \in W_{\mathcal{H}}$,

(*i*) $\mathcal{I}_{(\mathcal{H},w)} \models \forall E.\varphi$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \bigwedge_{i=1}^{n} [e_i/E]\varphi$;
(*ii*) $\mathcal{I}_{(\mathcal{H},w)} \models \exists E.\varphi$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \bigvee_{i=1}^{n} [e_i/E]\varphi$;
(*iii*) $\mathcal{I}_{(\mathcal{H},w)} \models \forall P.\varphi$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \bigwedge_{i=1}^{m} [p_i/P]\varphi$;
(*iv*) $\mathcal{I}_{(\mathcal{H},w)} \models \exists P.\varphi$ *iff* $\mathcal{I}_{(\mathcal{H},w)} \models \bigvee_{i=1}^{m} [p_i/P]\varphi$. ■

This property hints at the possibility of compiling a ***QCEC*** query to a formula that does not mention any quantifier. Observe however that this is possible only after an EC-structure has been specified. Therefore, quantifiers are not simply syntactic sugar, but an effective extension over a query language with connectives.

We will rely on the above lemma in order to analyze the computational complexity of the formalism in Section 5. However, we will not take advantage of it to implement ***QCEC*** in Section 4 since a model checker should be independent from the particular EC-problem it is operating on.

3 Example

In this section, we consider an example taken from the domain of medical diagnosis that shows how an extension of ***EC*** with quantifiers and connectives can be conveniently used to deal with significant applications.

We focus our attention on repeated clusters of events whose correlation, if present, can entail conclusions about the state of the system under observation. As an example, consider the following rule of thumb for diagnosing malaria [7]:

> ***A malaria attack begins with chills that are*** *followed by high* fever. ***Then the chills stop*** and some ***time later the*** fever goes away as well. Malaria ***is likely if the patient has repeated episodes*** *of* ***malaria attacks.***

Figure 1 describes the symptoms of a patient, Mr. Jones, who has just returned from a vacation to the Tropics. We have labeled the beginning and the end of chills and fever periods for reference. According to the rule above, Mr. Jones should be diagnosed with malaria. If however he had not had fever in the period between es and es for example, or if e_7 had preceded es, then further checks should be made in order to ascertain the kind of ailment he suffers from.

We will now show how the rule above can be expressed as a ***QCEC*** query in order to automate the diagnosis of malaria. The first task is to give a representation of symptom records as EC-problems. In the case of Mr. Jones, the factual information of his condition is represented by the EC-structure $\mathcal{H} = (\mathbf{E}, \mathbf{P}, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[)$ below, which is a direct transliteration of the data in Figure 1.

- $\mathbf{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$,
- **P = {chills, *fever*},**
- ***[chills)*** $= \{e_1, e_5, e_9\}$, $[fever\rangle = \{e_2, e_6, e_{10}\}$,
- ***(chills]*** $= \{e_3, e_7, e_{11}\}$, $\langle fever] = \{e_4, e_8, e_{12}\}$,
- $]\cdot,\cdot[= \emptyset$.

The events that initiate and terminate the symptoms of Mr. Jones happened in ascending order of their indices. We call w the corresponding ordering.

The decision rule for diagnosing malaria can then be reworded as saying that "whenever there is an episode ***of chills, there is a*** successive ***period of*** fever ***that starts before the chills*** are *over*".[1] It can in turn be expressed by the following formula in $\mathcal{L}_{\mathcal{H}}(QCEC)$:

$$\varphi = \forall E_1.\forall E_2.\,(chills(E_1, E_2) \supset (\exists E_1'.\,\exists E_2'.\,(E_1 < E_1' \text{ A } E_1' < E_2 \text{ A } fever(E_1', E_2'))))$$

that makes use of both universal and existential quantifiers over events, of all the connectives of QCEC (once implication is expanded) and of the precedence test. It is easy to verify that $\mathcal{I}_{(\mathcal{H},w)} \models \varphi$, while this formula is not valid in models where es or es have been eliminated, or where the relative order of es and e_7 has been reversed, for example.

There is no way to express this rule in ***EC,*** even when extended with connectives and the precedence test, unless quantifiers are unfolded as specified in Lemma 2.4. This would have however the undesirable effects of making the formula used to diagnose malaria problem-specific, and to augment considerably the size of this expression.

4 Implementation

The Event Calculus [5] has traditionally been implemented in the logic programming language Prolog [8]. Recent extensions to ***EC*** have instead adopted *λProlog* [6] in order to achieve a declarative yet simple encoding, necessary to formally establish correctness issues [2]. In this section, we will rely on orthogonal features of *λProlog* to obtain an elegant encoding of quantifiers (4.2). Before doing so, we recall the meaning of relevant constructs of this language (4.1). We conclude this section by showing that this program faithfully realizes the specification of ***QCEC (4.3).***

4.1 λProlog in a nutshell

Due to space limitations, we shall assume the reader to be familiar with the logic programming language ***Prolog*** [8]. We will instead illustrate some of the characteristic constructs of *λProlog* at an intuitive level. We invite the interested reader to consult [6] for a more complete discussion, and [2] for a presentation in the context of the Event Calculus.

Unlike ***Prolog*** which is first-order. *λProlog* is a ***higher-order*** language. which means that the terms in

[1] The other possible interpretations can be rendered in *QCEC*.

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
{*franzesc*|*montana*}*@dimi.uniud.it*

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 solv-

Appeared in the ***Proceedings of the Sixth International*** Conference on Principles ***of Knowledge*** Representation ***and Rea***soning — *KR'98* (A.G. Cohn, L.K. Schubert, and S.C. Shapiro, editors), pp. ??–??, Morgan Kaufmann Publishers, Trento, Italy, 2-5 June 1998.

ing *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.***
- $[\cdot\rangle : P \to 2^E$ *and* $\langle\cdot] : P \to 2^E$ ***are*** *respectively* ***the*** initiating *and* terminating map *of* $\mathcal{H}$. *For every property* $p \in P$, $[p\rangle$ ***and*** $\langle p]$ ***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.*** □

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 $p(e_i, e_t)$, where e_i and e_t are the events that respectively initiate and terminate the interval over which ***p*** holds maximally. Consequently, we adopt as the ***query language*** of ***EC*** the set $\mathcal{L}_{\mathcal{H}}(EC) = \{p(e_1, e_2) : p \in P$ and $e_1, e_2 \in E\}$ of all such property-labeled intervals over $\mathcal{H}$. We interpret the elements of $\mathcal{L}_{\mathcal{H}}(EC)$ as propositional letters and the task performed by ***EC*** reduces to deciding which of these formulas are MVIs in the current knowledge state w and which are not. This is a model checking problem.

In order for $p(e_1, e_2)$ to be an MVI relative to the event ordering w, it must be the case that $e_1 <_w e_2$. Moreover, e_1 and e_2 must witness the validity of the prop erty p at the ends of this interval by initiating and terminating ***p,*** respectively. The maximality requirement is caught by the negation of the meta-predicate $br(p, e_1, e_2, w)$ below. which expresses the fact that the validity of an MVI must not be *broken* by any interrupting event. Any event e which is known to have happened between e_1 and e_2 in w and that initiates

or terminates a property that is either ***p*** itself or a property exclusive with ***p*** interrupts the validity of $p(e_1, e_2)$ [4].

Definition ***2.2 (Intended model of EC)***

Let $\mathcal{H}$ ***= (E,*** $P, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[$) ***be a EC-structure. The*** intended EC-model ***of*** $\mathcal{H}$ ***is the propositional*** valuation $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \rightarrow 2^{\mathcal{L}_{\mathcal{H}}(EC)}$, ***where*** $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w)$ ***if and*** only ***if*** (i) $e_1 <_w e_2$, (ii) $e_1 \in [p\rangle$, (iii) $e_2 \in \langle p]$, (iv) $br(p, e_1, e_2, w)$ ***does not hold, where*** $br(p, e_1, e_2, w)$ ***abbreviates***

there exists an event $e \in E$ ***such that*** $e_1 <_w e$, $e <_w e_2$ ***and there exists a property*** $q \in P$ ***such that*** $e \in [q)$ ***or*** $e \in (q]$, ***and either*** $]p, q[$ ***or*** $p = q$. □

2.2 Modal EC with Connectives

The query language of the basic ***EC*** we just formalized suffers from a remarkably low expressive power that prevents its use for modelling any but the most trivial applications. The expressiveness of this formalism is drammatically augmented by admitting boolean connectives in queries. This allows inquiring about logical combinations of basic MVI verification problems.

In our specific setting, where the ordering of event occurrences is only partially specified, the set of MVIs computed by ***EC*** is not stable with respect to the acquisition of new ordering information. Indeed, as we move to an extension of the current knowledge state, some MVIs might become invalid and new MVIs can emerge [7]. Extending the query language of ***EC*** with the modal logic operators • I and 0 leads to the possibility of enquiring about which MVIs will remain valid in every extension of the current knowledge state, and about which intervals might become MVIs in some extension of it [1, 8]. Several ways of combining boolean connectives and modalities, with different cost and expressiveness, have been proposed [3, 5].

In this paper, we also include a precedence ***test*** operator, written $<$, which allows checking the relative order of two events in the current knowledge state. In previous work, this was awkwardly achieved either by augmenting EC-structures with dedicated properties [5], or by using preconditions [6]. A native precedence test makes inquiring about the relative order of two events independent from the underlying EC-structure.

Given an EC-structure $\mathcal{H}$, the query language that freely includes these three extensions is formally defined by the following grammar:

$$\varphi \ ::= \ p(e_1, e_2) \mid e_1 < e_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi.$$

We call this language $\mathcal{L}_{\mathcal{H}}(CMEC)$ and ***CMEC*** the relative extension of ***EC.*** In addition to the above operators, we admit implication as a derived connective, where $\varphi_1 \supset \varphi_2$ is classically defined as $\neg\varphi_1 \vee \varphi_2$.

In order to formalize the semantics of the modalities in ***CMEC,*** we must shift the focus from the current knowledge state w to all knowledge states that are reachable from w, i.e. $\mathrm{Ext}_{\mathcal{H}}(w)$. Since $\subseteq$ is a reflexive partial order, $(W_{\mathcal{H}}, \subseteq)$ can be naturally viewed as a finite, reflexive, transitive and antisymmetric modal frame. If we consider this frame together with the straightforward modal extension of the valuation $\upsilon_{\mathcal{H}}$ to an arbitrary knowledge state, we obtain a modal model for ***CMEC.*** Connectives are handled as usual and incorporating the precedence test is trivial.

Definition 2.3 ***(Intended model of*** $CMEC$***)***

Let $\mathcal{H}$ ***= (E,*** $P, [\cdot\rangle, \langle\cdot],]\cdot,\cdot[$) ***be an EC-structure. The*** intended CMEC-model ***of*** $\mathcal{H}$ ***is the modal model*** $\mathcal{I}_{\mathcal{H}} = (W_{\mathcal{H}}, \subseteq, \upsilon_{\mathcal{H}})$, ***where the propositional valuation*** $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \rightarrow 2^{\mathcal{L}_{\mathcal{H}}(EC)}$ ***is defined as*** in ***Definition 2.2. Given*** $w \in W_{\mathcal{H}}$ and $\varphi \in \mathcal{L}_{\mathcal{H}}(CMEC)$, ***the truth of*** φ ***at w with respect to*** $\mathcal{I}_{\mathcal{H}}$, ***denoted by*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi$, ***is defined as follows:***

$\mathcal{I}_{\mathcal{H}}; w \models p(e_1, e_2)$ ***iff*** $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w)$;
$\mathcal{I}_{\mathcal{H}}; w \models e_1 < e_2$ ***iff*** $e_1 <_w e_2$;
$\mathcal{I}_{\mathcal{H}}; w \models \neg\varphi$ ***iff*** $\mathcal{I}_{\mathcal{H}}; w \not\models \varphi$;
$\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \wedge \varphi_2$ ***iff*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi_1$ ***and*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi_2$;
$\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \vee \varphi_2$ ***iff*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi_1$ ***or*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi_2$;
$\mathcal{I}_{\mathcal{H}}; w \models \Box\varphi$ ***iff for all*** $w' \in \mathrm{Ext}_{\mathcal{H}}(w)$, $\mathcal{I}_{\mathcal{H}}; w' \models \varphi$;
$\mathcal{I}_{\mathcal{H}}; w \models \Diamond\varphi$ ***iff there is*** $w' \in \mathrm{Ext}_{\mathcal{H}}(w)$ ***such that*** $\mathcal{I}_{\mathcal{H}}; w' \models \varphi$. □

Notice that deciding the truth of a modal formula requires the exploration of all the extensions of the current knowledge state. Since there are exponentially many, this raises the complexity of ***CMEC*** beyond tractability [5]. This distressing fact is overcome in the calculus ***ECMEC*** [4, 5], that restricts ***CMEC*** by allowing □ and 0 to enclose only atomic formulas of the form $e_1 < e_2$ and $p(e_1, e_2)$. To determine the truth of atomic formulas prefixed by one modal operator, it is possible to exploit necessary and sufficient ***local conditions*** over the given partial order, thus avoiding a complete (and expensive) search of all the consistent extensions of the given order [5]. Therefore, solving modal queries in ***ECMEC*** has polynomial cost [5].

This is particularly appealing since numerous $CMEC$-formulas are logically equivalent to ECMEC-formulas. The transformation proceeds by pushing the modalities inside the scope of the connectives. An ***ECMEC*** formula cannot always be produced since □ does not

distribute over $\vee$, and dually **0** cannot be pushed inside a conjunction. We will now consider conditions that permit overcoming this difficulty in situations of interest.

Specifically, we consider EC-structures $\mathcal{H}$ = (E, P, $[\cdot\rangle$, $\langle\cdot]$, $]\cdot,\cdot[$)where every property is initiated and terminated by at most one event and there are no exclusive properties. We call this condition $(*)$. An atomic formula $p(e_1, e_2)$ on $\mathcal{H}$ is an MVI relative to the knowledge state w $\in W_{\mathcal{H}}$ if and only if e_1 initiates p, e_2 terminate p and (e_1, e_2) belongs to w. Indeed, condition $(*)$ ensures us that there are no interrupting events for p in (e_1, e_2) and thus we do not need to check whether $br(e_1, p, e_2, \text{w})$ holds since this meta-predicate will be trivially false. Condition $(*)$ offers further opportunities to push modalities inside the scope of connectives. We omit the proof of the following simple proposition.

Proposition 2.4 *(Consequences of* $(*)$*)*

Let $\mathcal{H}$ = ***(E,*** P, $[\cdot\rangle$, $\langle\cdot]$, $]\cdot,\cdot[$) ***be an EC-structure that satisfies*** $(*)$. ***Let*** φ ***be a*** CMEC-*formula. For* $p \in P$ ***and*** $e_1, e_2 \in E$, ***let*** $\nu_p(e_1, e_2)$ ***be either*** $p(e_1, e_2)$ ***or*** $(e_1 < e_2)$. ***Then, for any*** $w \in W_{\mathcal{H}}$ ***such that*** $e_1 <_w e_2$, ***we have that:***

i. $w \models \Box(\nu_p(e_1, e_2) \vee \varphi)$ *iff* $w \models \nu_p(e_1, e_2) \vee \Box\varphi$;
ii. $w \models \Diamond(\nu_p(e_1, e_2) \wedge \varphi)$ *iff* $w \models \nu_p(e_1, e_2) \wedge \Diamond\varphi$. ■

In particular, for φ = **false** (resp. **true), we** have that w $\models$ ❑ lv,(el, e_2) (resp. w $\models \Diamond\nu_p(e_1, e_2)$) *iff* $w \models \nu_p(e_1, e_2)$.

2.3 Modal EC with Connectives and Quantifiers

We will now enrich ***CMEC*** with explicit universal and existential event quantifiers that can be used freely in a query. We call the resulting formalism ***QCMEC. In***deed, a logic programming implementation of ***CMEC*** can emulate only restricted forms of existential quantification by means of unification, while universally quantified queries are out of reach.

In order to accommodate quantifiers, we extend the query language of an EC-structure $\mathcal{H}$ = (E, P, $[\cdot\rangle$, $\langle\cdot]$, $]\cdot,\cdot[$) in several respects. We first assume the existence of infinitely many ***event variables*** that we denote x, possibly subscripted. We write $\bar{e}$ for a syntactic entity that is either an event in E or an event variable. The query language of ***QCMEC,*** denoted $\mathcal{L}_{\mathcal{H}}(QCMEC)$, is the set of ***closed*** formulas generated by the following grammar:

$$\varphi ::= p(\bar{e}_1, \bar{e}_2) \mid \bar{e}_1 < \bar{e}_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \forall x.\varphi \mid \exists x.\varphi.$$

The notions of free and bound variables are defined as usual and we identify formulas that differ only by the name of their bound variables. We write $[e/x]\varphi$ for the substitution of an event e $\in E$ for every free occurrence of the event variable x in the formula φ. Notice that this limited form of substitution cannot lead to variable capture.

We now extend the notion of intended model to accommodate quantifiers.

Definition 2.5 *(Intended model of QCMEC)*

Let $\mathcal{H}$ = ***(E,*** P, $[\cdot\rangle$, $\langle\cdot]$, $]\cdot,\cdot[$) ***be*** an ***EC-structure. The*** intended QCMEC-model ***of*** $\mathcal{H}$ ***is the modal model*** $\mathcal{I}_{\mathcal{H}} = (W_{\mathcal{H}}, \subseteq, \upsilon_{\mathcal{H}})$ ***defined as in Definition 2.3. Given*** $w \in W_{\mathcal{H}}$ ***and a (closed) formula*** $\varphi \in \mathcal{L}_{\mathcal{H}}(QCMEC)$, ***the truth of*** φ ***at w with respect to*** $\mathcal{I}_{\mathcal{H}}$, ***denoted as*** $\mathcal{I}_{\mathcal{H}}; w \models \varphi$, ***is defined as*** in Definition 2.3 ***with the addition of the following two cases:***

$\mathcal{I}_{\mathcal{H}}; w \models \forall x.\varphi$ ***iff for all*** $e \in E$, $\mathcal{I}_{\mathcal{H}}; w \models [e/x]\varphi$;
$\mathcal{I}_{\mathcal{H}}; w \models 3x.\ \varphi$ ***iff there exists*** $e \in E$ ***such that*** $\mathcal{I}_{\mathcal{H}}; w \models [e/x]\varphi$. □

The well-foundedness of this definition derives from the observation that if $\forall x.\ \varphi$ and $\exists x.\ \varphi$ are closed formula, so is $[e/x]\varphi$ for every event e $\in E$.

A universal quantification over a finite domain can always be expanded into a finite sequence of conjunctions. Similarly an existentially quantified formula is equivalent to the disjunction of all its instances. The following lemma, whose simple proof we omit, applies these principles to ***QCMEC.***

Lemma 2.6 *(Unfolding quantifiers)*

Let $\mathcal{H}$ = ***(E,*** P, $[\cdot\rangle$, $\langle\cdot]$, $]\cdot,\cdot[$) ***be*** an EC-structure, ***with*** $E = \{e_1, \ldots, e_n\}$. Then, ***for every*** $w \in W_{\mathcal{H}}$,

i. $\mathcal{I}_{\mathcal{H}}; w \models \forall x.\ \varphi$ iff $\mathcal{I}_{\mathcal{H}}$; w $\models \bigwedge_{i=1}^{n} [e_i/x]\varphi$;
ii. $\mathcal{I}_{\mathcal{H}}; w \models \exists x.\ \varphi$ *iff* $\mathcal{I}_{\mathcal{H}}$; • $\models \bigvee_{i=1}^{n} [e_i/x]\varphi$. ■

This property hints at the possibility of compiling a ***QCMEC*** query to a quantifier-free formula. Observe however that this is possible only after an EC-structure has been specified. We will rely on the above lemma in order to analyze the explicit complexity of the formalism in Section 5. It is also possible to take advantage of it in order to structure an implementation of ***QCMEC*** into a preprocessor that expands quantifiers into exhaustive sets of conjunctions or disjunctions, and a $CMEC$ checker that verifies the resulting formula. We will however follow a more direct approach in Section 4.

We conclude this section by defining a quantified vari-

Part VI

Adaptive Agents

Diana Gordon: "Well-behaved Borgs, Bolos, and Berserkers", in the Proceedings of the 15th International Conference on Machine Learning — ICML'98, Morgan Kaufmann, Madison, WI, July 1998.

Full paper: `http://www.aic.nrl.navy.mil/~gordon/papers/ml98.ps`

This paper will appear in *the proceedings of the 15th International Conference on Machine Learning (ICML-98)*

Well-Behaved Borgs, Bolos, and Berserkers

Diana F. Gordon
Naval Research Laboratory, Code 5510
4555 Overlook Avenue, S.W.
Washington, DC 20375
gordon@aic.nrl.navy.mil

Abstract

How can ***we guarantee*** that our software and robotic agents will behave as we require, even after learning? Formal verification should play a key role but can be computationally expensive, particularly if re-verification follows each instance of learning. This is especially a problem if the agents need to make rapid decisions and learn quickly while online. Therefore, this paper presents novel methods for reducing the time complexity of re-verification subsequent to learning. The goal is agents that are predictable and can respond quickly to new situations.

1 INTRODUCTION

Software and robotic agents are becoming increasingly prevalent. Agent designers can furnish such agents with plans to perform desired tasks. Nevertheless, a designer cannot possibly foresee all circumstances that. will be encountered by the agent. Therefore, in addition to supplying an agent with plans, it is essential to also enable the agent to learn and modify its plans to adapt to unforeseen circumstances. The introduction of learning, on the other hand, often makes the agent's behavior significantly harder to predict. Our objective is to develop methods that provide verifiable guarantees that the behavior of learning agents always remains within the bounds of specified constraints (called "properties"), even after learning. An example of a property is Asimov's First Law of Robotics (Asimov, 1942). This law, which has recently been studied by Weld and Etzioni (1994), states that a robot may not harm a human or allow a human to come to harm. Weld and Etzioni advocate a " 'call to arms:' before we release autonomous agents into real-world environments, we need some credible and computationally tractable means of making them obey Asimov's First Law...how do we stop our artifacts from causing us harm in the process of obeying our orders?" Asimov's law can be operationalized into specific properties testable on a system, e.g., "Never delete another user's file." This paper addresses Weld and Etzioni's "call to arms" in the context of adaptive agents. It is a very important topic for real-world agents and is a dominant theme in science fiction, which is sometimes prescient. Examples include the Borgs (Star Trek, The New Generation), Bolos (Laumer, 1976), and Berserkers (Saberhagen, 1967) – fictional agents that demonstrate the dangerous behavior that can result from insufficient constraints.

We assume that an agent's plan has been initially verified offline. Then, the agent is fielded and has to adapt online. After adaptation via learning, the agent must rapidly re-verify its new plan to ensure this plan still satisfies required properties.' Re-verification must, be as computationally efficient as possible because it is performed online, perhaps in a highly time-critical situation. There are numerous applications of this scenario, including software agents that can safely access information in confidential or proprietary environments while responding to rapidly changing access requirements, planetary rovers that quickly adapt to unforeseen planetary conditions but behave within critical mission constraints, and JAVA applets that can get smarter but not become destructive to our computing environments.

Typically, properties desired by a user are orthogonal to the agent's planning goals and to its learning goals.

'Current output is success/failure. Future work will consider using re-verification counterexamples to choose a better learning method when re-verification fails.

For example, the agent may generate a plan with the objective of maximizing the agent's profit. Learning might have the goal of achieving the agent's plan more efficiently or modifying the plan to adapt to unforeseen events. The designer also may have a constraint that the agent does not cheat in its dealings with other agents. Why doesn't the planner incorporate all properties into the plan? There are a number of possible reasons, e.g., not all properties may be known at the time the plan is developed, or security reasons.

Re-verification can be (from least to most time required): none, incremental, or complete. It is possible to avoid re-verification entirely if we restrict the agent to using only those learning methods determined a prioii to be "safe" with respect to certain classes of properties in which we are interested. In other words, if a plan satisfies a property prior to learning, we want an a priori guarantee that the property will still be satisfied subsequent to learning. Note that this incurs no run-time cost. It is called "moving a tester into the generator" or "compiling constraints."

Unfortunately, the safety of some learning methods may be very difficult or maybe impossible to determine a priori. When a priori determination is too difficult, it is helpful to use incremental re-verification. Incremental methods save computational costs over re-verification from scratch by localizing re-verification and/or by reusing knowledge from the original verification. Furthermore, incremental methods may identify positive results that cannot be determined a priori. When an agent needs to learn, we suggest that the agent should consult the a priori results first. If no positive results exist, then incremental re-verification proceeds. The least desirable of the three alternatives is to do complete re-verification from scratch.

Gordon (1997a) begins to explore the extent to which we can prove a priori results that certain machine learning operators are, or are not, safe for certain classes of properties. The paper has positive a priori results for plan efficiency improvements via deletion of plan elements, as well as for plan refinement methods. Unfortunately, we have not yet obtained positive a priori results for popular machine learning operators such as abstraction (unless one is willing to accept an abstracted property) or generalization. Abstraction is a more global operator than generalization. Abstraction alters the language of a plan (e.g., by feature selection), whereas generalization alters the condition for a state-to-state transition within a plan. Both are extremely common operators in concept learning. but are also very appropriate for plan modification.

This paper has two contributions beyond (Gordon, 1997a). First, the previous paper models agent plans using automata on infinite strings. This paper reaches a wider audience by using the more familiar automata on finite strings. Second, this paper addresses two, new questions: Are there situations in which an abstracted property is acceptable? If yes, we have positive a priori results for abstraction. Also, can we get positive results by using incremental re-verification rather than a priori? Initial, positive answers to these questions are presented here.

The remainder of this paper is organized as follows. Section 2 presents an illustrative example that is used throughout the paper. [2] Section 3 contains background material and definitions on automaton plans, temporal logic properties, and "safe" learning. The formal definitions provide a precise foundation for understanding the incremental re-verification methods presented later. Section 4 lists situations in which property abstraction is acceptable. Sections 4 and 5 present novel (and as far as we are aware, the only) methods for incremental re-verification of abstraction and generalization, respectively, on automata. Finally, time complexity comparisons between incremental and complete re-verification are provided.

2 ILLUSTRATIVE EXAMPLE

This section provides an example to illustrate some of the main ideas of the paper. Although the plan in this example is very small, it is important to point out that existing automata-based verification methods currently handle huge, industrial-sized problems (e.g., see Kurshan, 1994). Our goal is to improve the time complexity of verification over current methods when learning occurs.

In our example, hundreds of tiny, micro air vehicles (MAVs) are required to perform a task within a region. The MAVs are divided into two groups called "swarm A" and "swarm B." One constraint, or property, is that only one MAV may enter the region at a time – because multiple MAVs entering simultaneously would increase the risk of detection. Each swarm has a separate FIFO queue of MAVs. MAVs enter the queue when they return from their last task. A second constraint is that some (at least one) MAVs from each swarm eventually enter the region. One distinguished MAV, C, acts as a

*Examples in this paper have been implemented using Kurshan's COSPAN verification system. COSPAN is an AT&T verification tool. which is described in Kurshan (1994).

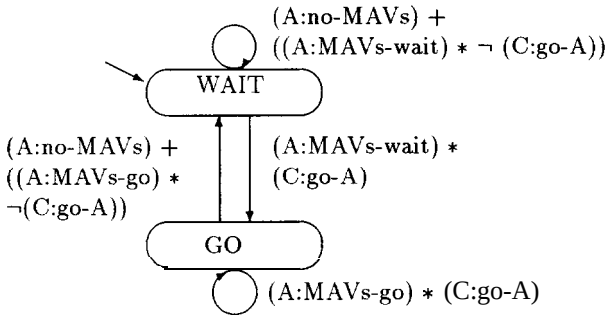

Figure 1: Plan A

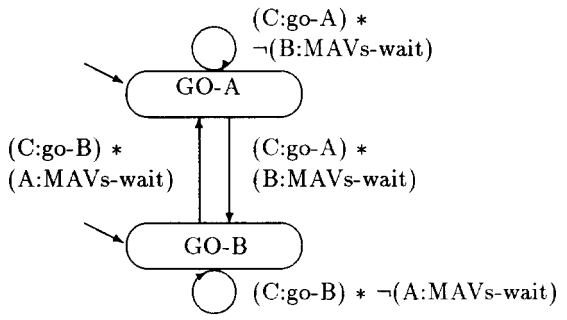

Figure 2: Plan C

task coordinator. C selects which swarm, A or B, may send in an MAV next.[3]

Plans for swarm A and task controller C are shown in Figures 1 and 2. The plan for swarm B is not shown in the figure, but it is identical to the plan for A except all instances of "A" are replaced by "B." Each of these plans is a finite-state automaton, i.e., a graph with states (the vertices) and allowable state-to-state transitions (the directed edges between vertices). The transition conditions (i.e., the logical expressions labeling the edges) describe the set of actions that enable a state transition to occur. The possible actions A can take from a state are (A:no-MAVs), (A:MAVs-wait), or (A:MAVs-go). The first action means the queue is empty, the second that the queue is not empty but the MAVs in the queue must wait, and the third that the first MAV in the queue enters the region. Likewise for B. The possible actions C can take from a state are (C:go-A) or (C:go-B). The first action means controller C allows swarm A to send one MAV into the region, the second means C allows B to send one MAV into the region.

Swarms A and B are single agents, i.e., although individual MAVs may each have their own plan, such as queuing within a swarm, for simplicity we ignore that level of detail. We can form a multiagent plan by taking a "product" (see Section 3.1) of the plans for A, B, and C. This product synchronizes the behavior of A, B, and C in a coordinated fashion. At every discrete time step, every agent (A, B, C) is at one state in its plan, and it selects its next action. The action of one agent (e.g., A) becomes an input to the other agents' plans (e.g., B and C). If the joint actions chosen by all three agents satisfy the transition conditions of a plan from the current state to some next state, then that transition may be made. For example, if the agents jointly take the actions (A:MAVs-wait) and (B:MAVs-wait) and (C:go-A), then the multiagent plan can transition from the global, joint state (WAIT, WAIT, GO-A) to the joint state (GO, WAIT, GO-B) represented by triples of states in the automata for agents A, B, and C.

Given the full, multiagent plan, verification now consists of asking the question: Does this plan satisfy the two required properties, i.e., some MAVs from each swarm enter the region, but only one MAV enters the region at a time? Assuming our initial plan in Figures 1 and 2 satisfies these properties, we next ask whether the properties are still satisfied subsequent to learning. The latter question is the topic of this paper.

An example of learning is the following. Suppose coordinator C discovers that the B swarm has left the region. One way agent C can adapt to incorporate this new knowledge is by deleting the action (C:go-B) from its action repertoire. This is a form of abstraction. There are alternative modifications agent C can do, but the selection between these alternatives is a learning issue, which we do not address here. What we do address here are the implications of this choice, in particular, which learning methods are safe, i.e., preserve the properties.

3 PLANS, PROPERTIES, AND "SAFE" LEARNING

3.1 AUTOMATON PLANS

This subsection, which is based on Kurshan (1994), briefly summarizes the basics of the automata used to model plans. Figures 1 and 2 illustrate the definitions. Essentially, an automaton is a graph with vertices corresponding to states and directed edges corresponding to state-to-state transitions. The terms "vertex" and "state" are used interchangeably throughout

[3] This example is a variant of the traffic controller in Kurshan (1994).

the paper. For an automaton representing an agent's plan, vertices represent the internal state of the agent and/or the state of its external environment. State-to-state transitions have associated transition conditions, which are the conditions under which the transition may be made. An agent action that satisfies a transition condition enables that transition to be made. We assume finite-state automata, i.e., the set of states is finite, and that the transition conditions are elements of a Boolean algebra. Therefore, we briefly diverge to summarize the basics of Boolean algebras.

A Boolean algebra $\mathcal{K}$ is a set with distinguished elements 0 and 1, closed under the Boolean operations $*$ (logical "and"), $+$ (logical "or"), and $\neg$ (logical negation), and satisfying the standard properties (Kurshan, 1994).

The Boolean algebras are assumed to be finite. There is a partial order among the elements, $\preceq$, which is defined as $x \preceq y$ if and only if $x * y = x$. The elements 0 and 1 are defined as Qx $\in \mathcal{K}$, 0 $\preceq$ x and Qx $\in \mathcal{K}$, x $\preceq$ 1. The ***atoms*** of $\mathcal{K}$, $\Gamma(\mathcal{K})$, are the nonzero elements of $\mathcal{K}$ minimal with respect, to $\preceq$. For two different atoms x and y within the ***same*** Boplean algebra, x $* y$ = 0. For Figures 1 and 2, agents A, B, and C each have their own Boolean algebra with its atoms. The atoms of A's Boolean algebra are the actions (A:no-MAVs), (A:MAVs-wait), and (A:bfAVs-go); the atoms of B's algebra are (B:no-MAVs), (B:MAVs-wait), and (B:MAVs-go); the atoms of C's algebra are (C:go-A) and (C:go-B).

A Boolean algebra $\mathcal{K}'$ is a ***subalgebra*** of $\mathcal{K}$ if $\mathcal{K}'$ is a non-empty subset of $\mathcal{K}$ that is closed under the operations $*$, +, and $\neg$, and also has the distinguished elements 0, 1. Let $\mathcal{K} = \prod \mathcal{K}_i$, i.e., $\mathcal{K}$ is the product algebra of the $\mathcal{K}_i$. In this case the $\mathcal{K}_i$ are subalgebras of $\mathcal{K}$. An atom of the product algebra is the product of the atoms of the subalgebras. For example, if $a_1, \ldots .$ a, are atoms of subalgebras $\mathcal{K}_1, \ldots . \mathcal{K}_n$, respectively, then $a_1 * \ldots * a,$ is an atom of $\mathcal{K}$.

In Figure 1, the Boolean algebra $\mathcal{A}$ used by agent A is the smallest one containing the atoms of A's algebra. It contains all Boolean elements formed from A's atoms using the Boolean operators $*$, $+$, and $\neg$, including 0 and 1. These same definitions hold for B and C's algebras $\mathcal{B}$ and C. One atom of the product algebra **ABC** is (A:no-MAVs) $*$ (B:no-MAVs) $*$ (C:go-A). This is the form of actions taken by the three agents in the multiagent plan. Algebras $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are subalgebra.5 of the product, algebra $\mathcal{ABC}$. Finally, $\mathcal{ABC}$ is the Boolean algebra for tile transit ion conditions in the multiagent plan.

Let us return now to automata. This paper focuses on automata that model agents with finite lifetimes (represented as a finite string, or sequence of actions). An example is an agent that is created specially to execute a plan and is destroyed immediately afterwards. In particular, we focus on processes. Processes are automata, but they are the dual of our usual notion of an automaton, which accepts any string beginning in an initial state and ending in a final state (Hopcroft & Ullman, 1979). Instead, processes accept any string beginning in an initial state and ending in a nonfinal state.[4] A string is a sequence of actions (atoms). Therefore, by specifying the set of final states, we can infer the set of action sequences not permitted by the plan. It consists of those strings ending in a final state. All other action sequences that begin in an initial state are permitted by the plan. Processes are used here to be consistent with the automata theoretic verification literature.

Formally, a process is a three-tuple S = $(M_{\mathcal{K}}(S), I(S), F(S))$ where $\mathcal{K}$ is the Boolean algebra corresponding to S. $M_{\mathcal{K}}(S)$: ***V(S) x V(S)*** $\rightarrow \mathcal{K}$ is the matrix of transition conditions, which are elements of $\mathcal{K}$, ***V(S)*** is the set of vertices of S, I(S) $\subseteq$ ***V(S)*** are the initial states, and F(S) $\subseteq$ ***V(S)*** are the final states. Also, E(S) = {e $\in$ ***V(S) x V(S)*** $\mid M_{\mathcal{K}}(e) \neq 0$} is the set of directed edges connecting pairs of vertices of S, and $M_{\mathcal{K}}(e)$ is the transition condition of $M_{\mathcal{K}}(S)$ corresponding to edge e. Note that we omit edges labeled "0." By our definition, an edge whose transition condition is 0 does not exist. We can alternatively denote $M_{\mathcal{K}}(e)$ as $M_{\mathcal{K}}(v_i, v_{i+1})$ for the transition condition corresponding to the edge going from vertex v_i to vertex v_{i+1}. For example, in Figure 1, $M_{\mathcal{K}}$ (WAIT, GO) is (A: MAVs-wait) $*$ (C: go-A).

Figures 1 and 2 illustrate the process definitions. There are process plans for two agents: swarm A and task coordinator C. Recall that agent B is identical to A but, with "A" replaced by "B." An incoming arrow to a state, not from any other state, signifies that this is an initial state. Recall that the output actions of process A are its atoms, and likewise for processes B and C. The transition conditions are the labels on the edges. We assume for process X = A, B, or C, F(X) .= 0, i.e., there are no final states. Therefore every finite string of actions that starts in an initial state and satisfies the transition conditions is accept-

[4] For the case of deterministic and complete transition conditions, reversing the acceptance condition will complement the language.