An Optimizing Compiler for Efficient Model Checking

Yifei Dong C.R. Ramakrishnan
ydong@cs.sunysb.edu cram@cs.sunysb.edu

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

Abstract

Different model checking tools offer a variety of specification languages to encode systems. These specifications are compiled into an intermediate form from which the global automata are derived at verification time. Some tools, such as SPIN, provide the user with constructs that can be used to affect the size of the global automata. In other tools, such as Murφ, the user specifies a system directly in terms of its global automata using a guarded command language, and hence has complete control over the automata sizes. Our experience shows that using low-level specifications we can significantly reduce verification times. The question then is, whether we can derive the low-level representations directly from a high-level specification without user intervention or dependence on user annotations.

We address this problem in this paper. We develop an optimizing compilation technique that transforms high-level specifications based on value-passing CCS into *rules* from which transitions of the global automata can be efficiently generated. The representation of rules is such that possible synchronizations can be computed at compile time in polynomial time while transitions can be generated during verification in unit time modulo indexing. We show, via experiments using examples with different characteristics, that our technique is very effective in practice. For example, the compiler reduces the verification time of our XMC model checker by a factor of up to fifteen, while reducing the space requirements by up to an order of magnitude. More importantly, we identify a set of optimizations that can be implemented with little compile-time overhead and significantly reduce the time and space required for verification.

Keywords: Model Checking, Specification Languages, Verification.

1 Introduction

Many powerful techniques and tools have been developed over the past decade for formally verifying properties of complex systems from their specifications (see [CW96b]). The specification languages supported by most tools have high-level features— such as data structures, procedures and parameterized processes— to reduce the chance of errors in the specifications themselves.

Model checking techniques [CE81, QS82, CES86], typically view the high-level system specification in terms of the underlying automaton: labeled transition system or Kripke structure [MP95]. Verification tools first *compile* the input specifications into a compact representation of the global transition relation. For example, SPIN [Hol97] compiles specifications in PROMELA into local automata, one for each process; CÆSAR [GS90] translates input LOTOS specifications [BB89] into Petri nets. At verification time, these representations are used by the explicit-state model checkers to construct the global automaton.

In contrast to the above tools, Murφ [Dil96] provides a guarded command language for specifying a system directly as rules defining its global transition relation. This permits the user to directly optimize the global automata by using several encoding tricks (such as grouping many individual computations and actions together into a single transition). A recent study of the performance of various verification tools (based on verifying the i-Protocol, a sliding window protocol in the GNU UUCP stack) shows that the low-level specification enables Murφ to be considerably faster, and consume much less memory, than other tools [DDR+99].

The immediate question that arises is whether we can achieve the verification efficiency of low-level specifications without sacrificing the ease of encoding and correctness of high-level specifications. Many tools, such as SPIN, let the user provide annotations that are then used to reduce the state space of global automata. For instance, using *atomic* and *d_step* constructs of PROMELA, a SPIN user can group many computation steps into one transition. However, the responsibility of assuring the correctness of the optimization lies solely with the user. Moreover, while these annotations reduce the state space to some extent, lower-level optimizations (as can be done in a Murφ specification) can reduce the state space even further.

An Optimizing Compiler for Model Checking: In this paper, we describe an optimizing compiler that automatically and efficiently translates high-level specifications (based on value-passing CCS [Mil89]) into a compact representation of the global automaton. We deploy several optimizations that reduce the state space of the global automaton *without relying on user annotations*. The salient features of the compilation technique are:

- The compiler translates value-passing CCS expressions into (Horn-clause like) rules that represent the transition relation of the global automaton. It precomputes possible synchronizations at compile-time while still maintaining polynomial (quadratic) bounds for compile time and space. The transition rules are represented such that transitions of the global automaton can be computed in unit time (modulo indexing) during verification.

- The compiler is derived directly from the SOS semantics of value-passing CCS expressions. This is in contrast to the translation for LOTOS programs described in [GS90], where the target Petri nets are derived from specifications other than LOTOS operational semantics. However, similar to [GS90], our compiler translates all valid expressions except those with recursion over parallel composition and relabeling operations.

- Consecutive computation steps are grouped into single atomic transitions *even across basic block boundaries*, leading to significant reductions in the state space. Note that in SPIN, in contrast, we can group together only those computations that span whole blocks using *atomic* annotations. Experiments show that our aggressive optimization reduces the state space by more than an order of magnitude in some examples, with consequent improvement in verification performance.

- The compiler takes time that is polynomial in the size of the input specification. Note that while techniques based on bisimulation equivalence can be used to *minimize* the global automata, they take time proportional to the state space of the system, and hence can be exponential.

Our Approach and its Effectiveness: We describe the compilation technique based on the XMC system [RRR+97], a model checker for modal mu-calculus [Koz83] and XL, a process description language that extends the value-passing CCS with parameterized processes, sequential composition and procedure calls. The XMC system is built using the XSB tabled logic programming system [XSB98]. The core of the system consists of two predicates, `trans`:$State \times Action \times State$ which computes the transition relation by interpreting XL process terms, and `models`:$State \times Formula$ which verifies whether a state represented by a process term models a given modal mu-calculus formula. The two predicates directly encode the structural operational semantics of XL and modal mu-calculus respectively. The encoding exploits the capabilities of the XSB logic programming system to compute least fixed points very efficiently. Greatest fixed point computations are encoded as negation of their dual least fixed point.

This encoding reduces model checking to logic program query evaluation. The XSB logic programming system performs goal-directed evaluation of queries, and consequently, we obtain a local ("on-the-fly") model checker. This high-level encoding comes with little performance penalty. Using XMC, we can verify systems with more than a hundred thousand reachable states in times that are comparable to SPIN and Murφ. A detailed description of the XMC system can be found in [CDD+98].

We implemented the compiler in the XMC system and evaluated its effectiveness using benchmark programs with different characteristics. The compiler speeds up the XMC model checker by factors of up to 15, while reducing space requirements by factors of 6 or more. Using the compiler we can verify high-level specifications of several examples in the XMC system as fast as verifying equivalent low-level specifications in Murφ. For instance, XMC with compilation is more than 4 times faster than XMC without compilation for verifying the i-Protocol, and nearly as fast as Murφ. For the same example, verification using XMC with compilation is sometimes more than an order of magnitude faster than that using SPIN, and never slower.

Organization: We first present the structural operational semantics of XL, which forms the basis of our compiler (Section 2). We then formally describe a continuation-passing-style [App92] optimizing compiler for XL (Section 3). Experimental results measuring the effectiveness of the compilation technique and the various optimizations are presented in Section 4. Although designed for XL and the XMC system, several features of the compilation technique and the optimizations can be incorporated into the translation of any high-level specification language. We discuss these issues in Section 5.

2 The Specification Language XL

XL is based on the value passing CCS [Mil89]. Values and computations are represented as Prolog terms and predicates respectively. Thus the specifications can make use of recursive data structures and computations.

The syntax of XL specifications are described by the grammar shown in Figure 1. In the figure, *Proc* is a (parameterized) process name, represented as a term (e.g., `channel(N, Buf)`). *Comp* is a term (*e.g.,* `X is Y+1`) representing a computation. A terminating null process is represented by `true`, the empty computation. A process `if(`C`,`S_1`,`S_2`)`, behaves like S_1 if computation C succeeds, and like S_2 if C fails. The computation C in an `if` expression is assumed to leave the bindings of variables unchanged. A process `in(chan(`t`))` inputs a value

$$
\begin{array}{lcll}
E & \longrightarrow & \mathit{Comp} & \text{(computation)} \\
 & \mid & \texttt{in(}\mathit{Term}\texttt{)} & \text{(input communication)} \\
 & \mid & \texttt{out(}\mathit{Term}\texttt{)} & \text{(output communication)} \\
 & \mid & \texttt{zero} & \text{(unit deadlocked process)} \\
 & \mid & E \ \texttt{o} \ E & \text{(sequential composition)} \\
 & \mid & E \ \texttt{\#} \ E & \text{(choice)} \\
 & \mid & \texttt{if(}\mathit{Comp}\texttt{, } E\texttt{, } E\texttt{)} & \text{(conditional)} \\
 & \mid & E \ \texttt{|} \ E & \text{(parallel composition)} \\
 & \mid & E \setminus \mathit{PortSet} & \text{(restriction)} \\
 & \mid & E \ \texttt{@} \ \mathit{PortMap} & \text{(relabeling)} \\
 & \mid & \mathit{Proc} & \text{(process invocation)} \\
\mathit{Def} & \longrightarrow & (\mathit{Proc} \ \texttt{::=} \ E)^* & \text{(process definitions)}
\end{array}
$$

Figure 1: Syntax of XL

.

```
channel ::= in(get(Data)) o ( out(put(Data)) # out(drop) ) o channel.

sender(Seq) ::=
% Seq is the sequence number of the next frame to be sent
    out(dataOut(Seq)) o
    (   ( in(ackIn(AckSeq)) o
          if( AckSeq == Seq
              , NSeq is 1-Seq o sender(NSeq)          % successful ack, next message
              , sender(Seq))                          % unexpected ack, resend message
        ) # sender(Seq)).                             % upon timeout, resend message

receiver(Seq) ::=
% Seq is the expected sequence number of the next frame to be received
    in(dataIn(RecSeq)) o
    if( RecSeq == Seq
        , ( NSeq is 1-Seq o out(ackOut(RecSeq)) o receiver(NSeq) )
        , out(ackOut(RecSeq)) o receiver(Seq)).         % unexpected seq, resend ack

abp ::=
    ( sender(0) @ [s2r_in(X) / dataOut(X), r2s_out(X) / ackIn(X)]
    | channel @ [s2r_in(X) / get(X), s2r_out(X) / put(X)]     % sender -> receiver
    | channel @ [r2s_in(X) / get(X), r2s_out(X) / put(X)]     % receiver -> sender
    | receiver(0) @ [s2r_out(X) / dataIn(X), r2s_in(X) / ackOut(X)]
    ) \ {s2r_in(_), s2r_out(_), r2s_in(_), r2s_out(_)}.
```

Figure 2: Specification of the Alternating Bit Protocol in XL

that matches term t over port `chan`; `out(chan(`t`))` outputs the value represented by term t over port `chan`. Process invocations may be recursive; in fact, since the language provides no iterative constructs, recursion is the only way to specify loops in processes. As in CCS, relabeling and restriction are used to derive instances of a generic process. *PortSet* is a set of terms that represent the restricted ports, and *PortMap* is a list of pairs of terms (each pair of the form s/t) that denotes the mapping defined by relabeling.

The complete specification of the Alternating Bit Protocol [Tan96] in Figure 2 illustrates the features of XL. Observe from the figure that computation and communication are freely intermixed in an XL specification. Moreover, there are no user annotations that mark which computations can be treated as atomic: these are inferred automatically by the compiler.

$$(1)\quad \frac{}{\texttt{in}(t),\theta \xrightarrow{\texttt{in}(t\theta)} \texttt{true},\theta} \qquad (2)\quad \frac{}{\texttt{out}(t),\theta \xrightarrow{\texttt{out}(t\theta)} \texttt{true},\theta}$$

$$(3)\quad \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \circ F,\theta \xrightarrow{\alpha} E' \circ F,\theta'} \qquad (4)\quad \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{\texttt{true} \circ E,\theta \xrightarrow{\alpha} E',\theta'}$$

$$(5)\quad \frac{E_1,\theta \xrightarrow{\alpha} E_1',\theta'}{E_1 \ \texttt{\#}\ E_2,\theta \xrightarrow{\alpha} E_1',\theta'} \qquad (6)\quad \frac{E_2,\theta \xrightarrow{\alpha} E_2',\theta'}{E_1 \ \texttt{\#}\ E_2,\theta \xrightarrow{\alpha} E_2',\theta'}$$

$$(7)\quad \frac{[\![C]\!]_L\theta \neq \{\} \ \wedge \ E_1,\theta \xrightarrow{\alpha} E_1',\theta'}{\texttt{if}(C\texttt{,}\ E_1\texttt{,}\ E_2\texttt{)},\theta \xrightarrow{\alpha} E_1',\theta'}$$

$$(8)\quad \frac{[\![\texttt{not}(C)]\!]_L\theta \neq \{\} \ \wedge \ E_2,\theta \xrightarrow{\alpha} E_2',\theta'}{\texttt{if}(C\texttt{,}\ E_1\texttt{,}\ E_2\texttt{)},\theta \xrightarrow{\alpha} E_2',\theta'}$$

$$(9)\quad \frac{E_1,\theta \xrightarrow{\alpha} E_1',\theta'}{E_1 \mid E_2,\theta \xrightarrow{\alpha} E_1' \mid E_2,\theta'} \qquad (10)\quad \frac{E_2,\theta \xrightarrow{\alpha} E_2',\theta'}{E_1 \mid E_2,\theta \xrightarrow{\alpha} E_1 \mid E_2',\theta'}$$

$$(11)\quad \frac{E_1,\theta_1 \xrightarrow{\alpha} E_1',\theta_1' \ \wedge \ E_2,\theta_2 \xrightarrow{\beta} E_2',\theta_2' \ \wedge \ \{\alpha,\beta\} \equiv \{\texttt{in}(t),\texttt{out}(t\sigma)\}}{E_1 \mid E_2,\ \theta_1\theta_2\sigma \xrightarrow{\texttt{tau}} E_1' \mid E_2',\ \theta_1'\theta_2'\sigma}$$

$$(12)\quad \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{P',\sigma\theta \xrightarrow{\alpha} E',\sigma\theta'} \quad (P\texttt{::=}\ E, \sigma = mgu(P,P')) \qquad (13)\quad \frac{[\![Comp]\!]_L\theta = \theta'}{Comp,\theta \xrightarrow{\texttt{i}} \texttt{true},\theta'}$$

$$(14)\quad \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \setminus L,\theta \xrightarrow{\alpha} E' \setminus L,\theta'} \quad (\alpha \notin L) \qquad (15)\quad \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \ \texttt{@}\ F,\theta \xrightarrow{F(\alpha)} E' \ \texttt{@}\ F,\theta'}$$

Figure 3: Operational Semantics of XL

2.1 Operational Semantics of XL

In the following, we assume familiarity with the notions of terms and substitutions. The *unifier* of two terms t_1 and t_2 is a substitution θ such that $t_1\theta = t_2\theta$. The most general unifier of t_1 and t_2 is denoted by $mgu(t_1, t_2)$.

Following CCS, the structural operational semantics of XL can be specified in terms of labeled transition systems. In Figure 3 the semantics of an XL process is specified in terms of transitions in a labeled transition system where each state is represented by a process/substitution pair E, θ. We use $E,\theta \xrightarrow{\alpha} E',\theta'$ to denote that a process E under substitution θ can make a transition with label α to become process E' under substitution θ'. For computation C, $[\![C]\!]_L\theta$ is defined as the substitution θ' that is the result of evaluating the Prolog query $C\theta$. We say that $[\![C]\!]_L\theta = \{\}$ if the query $C\theta$ fails.

The semantics of communication primitives is specified by rules 1 and 2, sequential composition by rules 3 and 4, choice by rules 5 and 6, and the conditional by rules 7 and 8. Rules 9 and 10 denote autonomous transitions in a parallel composition. Rule 11 captures synchronization by matching the transition labels of the two components. Values transmitted by synchronizing `in`(t) with `out`(t') are represented by the matching substitution σ (i.e., σ such that $t' = t\sigma$).

Process invocation is represented by rule 12: when a process P' is invoked under substitution θ, a definition $P{:}{:}{=}E$ can be chosen such that P and P' unify, with the most general unifier composed with the current substitution θ. As is normal in such cases, we assume that P: :=E and P' are standardized apart before the unifier is computed: i.e., variables in P and E are suitably renamed to avoid capture.

Rule 13 captures the semantics of *computation* in XL in terms of the labeled transition system: the system makes an internal transition labeled `i`, changing the substitution as directed by the computation. The `i`-transitions which arise due to computations are considered distinct from `tau`-transitions which arise from synchronization.

Rule 14 specifies the semantics of restriction; the rule is made identical to that of basic CCS by overloading the meaning of membership '$\in$' as follows: Given an list L of terms, $s \in L$ if there is a term $t \in L$ such that $s =$`in(`t`)` or $s =$`out(`t`)`. Similarly, rule 15 uses a relabeling function as follows: given a list F of pairs of terms, if s/t is a member of F then F(`in(`t`)`) $=$`in(`s`)` and F(`out(`t`)`) $=$`out(`s`)`; if there is no s such that s/t is a member of F then $F(t) = t$.

3 Compiling XL

The operational semantics of XL defines the transition relation of the automaton corresponding to an XL specification. Given an XL program, the objective of the compiler is to derive a set of rules that precisely and concisely describes the corresponding transition relation. The conciseness requirement is especially important since the transition relation may be exponential in the size of the input program.

We follow the traditional approach for concisely representing large (even infinite) automata: separating *control* from *data*. A (possibly infinite) set S of states of an automaton is characterized by a single *control state* π, which is analogous to a program counter value. Associated with each control state π are a set of variables, denoted by $vars(\pi)$. For convenience, we use a term that contains $vars(\pi)$ to represent the control state π itself. Each state in the labeled transition system can then be represented by $\pi\sigma$: a control state π under a substitution σ. Transitions between states $\pi_s\sigma_s$ and $\pi_d\sigma_d$ can be captured by transition rules of the form `trans(`π_s`, `α`, `π_d`, `c`)`, where α is the label and c relates σ_s and σ_d. The control automaton closely mirrors XL's operational semantics. Given a process expression E, control states in the automaton correspond to the subexpressions of E, and the variables at any control state is a subset of variables of E.

3.1 Basic Compilation of XL

We compile each process expression into an automaton with two distinguished control states: *entry* and *exit*. The compiler is described in Figure 4 in terms the semantic function $[\![E]\!]$ *entry exit* which maps the expression E to a set of the transition rules (`trans`). In the figure, we use $\langle a_1, a_2, \ldots, \rangle$ to denote sequences and $\oplus$ to denote concatenation of sequences.

Compiling Process Invocations: Each process itself is compiled into a set of `trans` rules with a distinguished entry point. The control and data flow associated with calls and returns are uniformly handled by passing the current continuation as the first argument to the called process (the term *exit* in equation 10a and corresponding parameter *Cont* in equation 11) and "jumping" to the continuation at the end of a process (equation 11). The continuation-passing style naturally limits the number of `i`-transitions introduced by the compiler. Pa-

$$[\![\mathit{Comp}]\!]\ \mathit{entry}\ \mathit{exit} = \{\ \texttt{trans(}\mathit{entry}\texttt{, i, }\mathit{exit}\texttt{, }\mathit{Comp}\texttt{)}\ \} \qquad \cdots (1)$$

$$[\![\texttt{in(}t\texttt{)}]\!]\ \mathit{entry}\ \mathit{exit} = \{\ \texttt{trans(}\mathit{entry}\texttt{, in(}t\texttt{), }\mathit{exit}\texttt{, true)}\ \} \qquad \cdots (2)$$

$$[\![\texttt{out(}t\texttt{)}]\!]\ \mathit{entry}\ \mathit{exit} = \{\ \texttt{trans(}\mathit{entry}\texttt{, out(}t\texttt{), }\mathit{exit}\texttt{, true)}\ \} \qquad \cdots (3)$$

$$[\![E_1\ \texttt{o}\ E_2]\!]\ \mathit{entry}\ \mathit{exit} = [\![E_1]\!]\ \mathit{entry}\ \mathit{mid}\ \cup [\![E_2]\!]\ \mathit{mid}\ \mathit{exit} \qquad \cdots (4)$$

where *mid* is a new control state such that
$\mathit{vars}(\mathit{mid}) = (\mathit{vars}(\mathit{entry}) \cup \mathit{vars}(E_1)) \cap (\mathit{vars}(\mathit{exit}) \cup \mathit{vars}(E_2))$.

$$[\![E_1\ \texttt{\#}\ E_2]\!]\ \mathit{entry}\ \mathit{exit} = [\![E_1]\!]\ \mathit{entry}\ \mathit{exit}\ \cup [\![E_2]\!]\ \mathit{entry}\ \mathit{exit} \qquad \cdots (5)$$

$$\begin{aligned}[\![\texttt{if(}C, E_1, E_2\texttt{)}]\!]\ \mathit{entry}\ \mathit{exit} = {} & \{\ \texttt{trans(}\mathit{entry}\texttt{, i, }\mathit{iftrue}\texttt{, }C\texttt{)}, \\ & \quad \texttt{trans(}\mathit{entry}\texttt{, i, }\mathit{iffalse}\texttt{, not(}C\texttt{))}\ \} && \cdots (6a) \\ & \cup [\![E_1]\!]\ \mathit{iftrue}\ \mathit{exit} && \cdots (6b) \\ & \cup [\![E_2]\!]\ \mathit{iffalse}\ \mathit{exit} && \cdots (6c)\end{aligned}$$

where *iftrue* and *iffalse* are new control states such that
$\mathit{vars}(\mathit{iftrue}) = (\mathit{vars}(\mathit{entry}) \cup \mathit{vars}(C)) \cap (\mathit{vars}(\mathit{exit}) \cup \mathit{vars}(E_1))$
$\mathit{vars}(\mathit{iffalse}) = (\mathit{vars}(\mathit{entry}) \cup \mathit{vars}(C)) \cap (\mathit{vars}(\mathit{exit}) \cup \mathit{vars}(E_2))$

$$\begin{aligned}[\![E_1\ \ |\ \ E_2]\!]\ \mathit{entry}\ \mathit{exit} = {} & \{\ \texttt{trans(}s_1 \times V_2\texttt{, }\alpha_1\texttt{, }d_1 \times V_2\texttt{, }c_1\texttt{)}\ \text{such that} && \cdots (7a) \\ & \quad \texttt{trans(}s_1\texttt{,}\alpha_1\texttt{,}d_1\texttt{,}c_1\texttt{)} \in [\![E_1]\!]\ \mathit{entry}\ \mathit{exit} \\ & \quad \text{and } V_2 \text{ is a fresh variable}\} \\ & \cup \{\ \texttt{trans(}V_1 \times s_2\texttt{, }\alpha_2\texttt{, }V_1 \times d_2\texttt{, }c_2\texttt{)}\ \text{such that} && \cdots (7b) \\ & \quad \text{such that } \texttt{trans(}s_2\texttt{,}\alpha_2\texttt{,}d_2\texttt{,}c_2\texttt{)} \in [\![E_2]\!]\ \mathit{entry}\ \mathit{exit} \\ & \quad \text{and } V_1 \text{ is a fresh variable}\} \\ & \cup \{\ \texttt{trans(}(s_1 \times s_2)\theta\texttt{, tau, }(d_1 \times d_2)\theta\texttt{, }(c_1, c_2)\theta\ \texttt{)}\ \text{such that} && \cdots (7c) \\ & \quad \texttt{trans(}s_1\texttt{,}\alpha_1\texttt{,}d_1\texttt{,}c_1\texttt{)} \in [\![E_1]\!]\ \mathit{entry}\ \mathit{exit} \\ & \quad \wedge\ \texttt{trans(}s_2\texttt{,}\alpha_2\texttt{,}d_2\texttt{,}c_2\texttt{)} \in [\![E_2]\!]\ \mathit{entry}\ \mathit{exit} \\ & \quad \text{and } \theta \text{ is the most general substitution such that} \\ & \quad \{\alpha_1, \alpha_2\} \equiv \{\texttt{in(}t\texttt{)}, \texttt{out(}t\theta\texttt{)}\} \text{ for some term } t\ \}.\end{aligned}$$

$$\begin{aligned}[\![E\ \setminus\ L]\!]\ \mathit{entry}\ \mathit{exit} = {} & \{\ \texttt{trans(}\rho_{E,L}(s)\texttt{, }\alpha\texttt{, }\rho_{E,L}(d)\texttt{, }c\texttt{)}\ \text{such that} && \cdots (8) \\ & \quad \texttt{trans(}s\texttt{,}\alpha\texttt{,}d\texttt{,}c\texttt{)} \in [\![E]\!]\ \mathit{entry}\ \mathit{exit}\ \wedge\ \alpha \notin L\ \}\end{aligned}$$

where $\rho_{E,L}$ maps every control state except *entry* and *exit* to a new control state.

$$\begin{aligned}[\![E\ \texttt{@}\ F]\!]\ \mathit{entry}\ \mathit{exit} = {} & \{\ \texttt{trans(}\rho_{E,F}(s)\texttt{, }F(\alpha)\texttt{, }\rho_{E,F}(d)\texttt{, }c\texttt{)}\ \text{such that} && \cdots (9) \\ & \quad \texttt{trans(}s\texttt{,}\alpha\texttt{,}d\texttt{,}c\texttt{)} \in [\![E]\!]\ \mathit{entry}\ \mathit{exit}\ \}\end{aligned}$$

where $\rho_{E,F}$ maps every control state except *entry* and *exit* to a new control state.

$$\begin{aligned}[\![\mathit{Proc}]\!]\ \mathit{entry}\ \mathit{exit} = {} & \{\ \texttt{trans(}\mathit{entry}\texttt{, i,} && \cdots (10a) \\ & \qquad \mathit{entry\_point}(\mathit{Proc})(\langle \mathit{exit} \rangle \oplus \mathit{subterms}(\mathit{Proc}))\texttt{, true)}\ \} \\ & \cup [\![\mathit{definitions}(\mathit{Proc})]\!] && \cdots (10b)\end{aligned}$$

where *entry_point*(*Proc*) is the name of the start control state of the automaton corresponding to *Proc*, *subterms*(t) is the sequence of immediate subterms of term t, and *definitions*(P) is the set of all definitions P'`::=` E' in the input specification such that P unifies with P'.

$$[\![\mathit{Proc}\texttt{::=}\ E]\!] = [\![E]\!]\ \mathit{entry\_point}(\mathit{Proc})(\langle \mathit{Cont} \rangle \oplus \mathit{subterms}(\mathit{Proc}))\ \ \mathit{Cont} \qquad \cdots (11)$$

where *entry* and *exit* are new control states, *entry_point* and *subterms* are as defined in the previous rule and *Cont* is a fresh variable.

Figure 4: Compilation Rules for XL

rameter passing and variable renaming are delegated to the underlying Logic Programming engine as follows. We encode the caller state with the arguments, i.e., subterms in the call, (equation 10a). We then access the parameters in the callee using the subterms in the pattern defining the callee (equation 11). Note that for each process expression E, the compilation rules associate the set of *all* `trans` rules representing the transitions of E. If E contains a process invocation, say P', then $[\![E]\!]$ contains the `trans` rules corresponding to the process P' as well (due to equation 10b). This convention considerably simplifies the compilation of expressions with parallel, restriction and relabeling operations.

The first six equations in Figure 4 are direct encodings of the corresponding semantic rules (1–8, 13) in Figure 3. It should be noted that, even though we associate with each E all transition rules for each sub-automaton for E, the potential blow up is avoided by keeping the transition rules as *sets* and using set union to combine the rules from expressions with conditional, choice and sequential composition operations.

Compiling Parallel Composition: If E_1 and E_2 have automata with control states drawn from S_1 and S_2 respectively, the automaton corresponding to E_1 `|` E_2 has control states drawn from $S_1 \times S_2$. Equations 7a and 7b correspond to E_1 and E_2 making autonomous moves respectively. Equation 7c captures synchronizing transitions. While at first sight it appears as though precomputing the synchronizations will result in exponential blow up of the number of transition rules generated, the following argument shows otherwise. In the compilation of E_1 `|` E_2, let the automata for E_1 and E_2 have n_1 and n_2 non-`tau` transitions and m_1 and m_2 `tau` transitions respectively. The automaton constructed for E_1 `|` E_2 using equation 7 has $n_1 + n_2$ non-`tau` transitions (from equations 7a and 7b); $m_1 + m_2$ (from 7a and 7b) $+n_1n_2$ (from 7c) `tau` transitions. Note that the product operation in 7c takes only non-`tau` transitions and produces only `tau` transitions. Hence, the results of the product cannot be fed into a product operation again, thereby averting the exponential blow-up. In fact, a rough analysis of the product operation yields an upper bound of n^2N^2 transitions in a parallel composition of N automata with at most n transitions each. However, we find that the number of transition rules generated in practice is much smaller than the above bound. For instance, the number of transition rules for leader election protocol as well as sieve of Eratosthenes grow linearly with the number of processes.

An expression with restriction $E\backslash L$ is compiled by discarding any transitions with labels in L from a fresh copy of the automaton for E. Similarly, an expression with relabeling E `@` F is compiled by suitably renaming all transitions in a fresh copy of the automaton for E. Note that by parameterizing the "copying" function ρ with respect to E and L (or E and F), we avoid making multiple copies of the automaton for E if there are multiple occurrences of $E\backslash L$ (or E `@` F) in any expression.

Soundness and Termination of Compilation: The soundness of the compilation follows from the relationship between the compilation rules and the inference rules of XL's operational semantics. Note that the transition rules generated by the compiler may create more `i`-transitions. For instance, when compiling `if(`C`,`E_1`,`E_2`)`, the compiler inserts `i`-transitions between testing C and evaluating E_1 or E_2. However, since the substitutions of variables remains unaffected, the interleaving semantics of the generated transition rules coincides with the one given in Figure 3.

The compilation process can be implemented in the XSB tabled logic programming system [XSB98] by simply encoding the (set) equations in Figure 4 as a Horn clause program and evaluating it using tabled resolution [CW96a]. Termination of such a compiler can be readily shown by induction on the structure of process expressions, whenever there is no parallel composition within a recursive process definition. An XL specification that overlaps parallel composition with recursion, (e.g., `p(s(N)) ::= q | p(N)`) cannot be handled by the compilation scheme. While the scheme can be extended to compile parameterized processes

```
trans(abp(A,channel_1(B,C),D,E), out(drop), abp(A,channel_0(C),D,E), true).
trans(abp(A,B,channel_1(C,D),E), out(drop), abp(A,B,channel_0(D),E), true).
trans(abp(sender_1(A,B),C,D,E), i, abp(sender_0(A,B),C,D,E), true).
trans(abp(sender_0(A,B),channel_0(C),D,E), tau, abp(sender_1(A,B),channel_1(A,C),D,E), true).
trans(abp(sender_1(A,B),C,channel_1(D,E),F), tau, abp(sender_0(G,B),C,channel_0(E),F),
    (D == A,  G is 1 - A)).
trans(abp(sender_1(A,B),C,channel_1(D,E),F), tau, abp(sender_0(A,B),C,channel_0(E),F),
    (not(D == A))).
trans(abp(A,channel_1(B,C),D,receiver_0(E,F)), tau, abp(A,channel_0(C),D,receiver_4(B,G,F)),
    (B == E,  G is 1 - E)).
trans(abp(A,channel_1(B,C),D,receiver_0(E,F)), tau, abp(A,channel_0(C),D,receiver_3(B,E,F)),
    (not(B == E))).
trans(abp(A,B,channel_0(C),receiver_4(D,E,F)), tau,
        abp(A,B,channel_1(D,C),receiver_0(E,F)), true).
trans(abp(A,B,channel_0(C),receiver_3(D,E,F)), tau,
        abp(A,B,channel_1(D,C),receiver_0(E,F)), true).
```

Figure 5: Transition Rules for the Alternating Bit Protocol

(such as `p(N)` above) whenever the value of the parameter is known at compile time. However, extending the compilation scheme to *dynamically* compile process expressions whenever processes are created at verification time remains an interesting open problem.

3.2 Optimizations

The XL compiler performs several optimizations, described below. To illustrate compilation and optimization, we show in Figure 5 the transition rules for process `abp` of the alternating bit protocol (Figure 2) derived using our compiler.

Reducing Internal (`i`-) Transitions: Internal transitions are generated by the compiler for computations, conditional evaluation and process invocation. Among these, consider the `i`-transitions generated for computations. Different interleavings of these computations among concurrent processes cannot be distinguished by any modal mu-calculus formula as long as the computations do not affect the bindings of shared variables. Hence, such `i`-transitions can be treated like ϵ-transitions and eliminated from the rule set wherever possible, thereby reducing the states and the number of interleavings considered at verification time. Note that the empty computation, `true`, is associated with `i`-transitions generated for conditional and process invocation expressions; hence these too can be treated as ϵ-transitions.

Every transition rule of the form `trans(`s`, i, `s'`, `c`)` where c affects only local variables can be replaced with the set of transition rules of the form `trans(`s`, `α_i`, `t_i`, `(c, c_i)`)` where `trans(`s'`, `α_i`, `t_i`, `c_i`)` are the set of transition rules from control state s'. Note that such a transformation can potentially increase the size of the rule set exponentially; however, the benefits of the optimization appears to outweigh the potential (although rare) blow ups.

In addition, a transition rule of the form `trans(`s`, `α`, `s'`, `c`)` where the only transition rule from state s' is of the form `trans(`s'`, i, `t`, `c'`)` such that c' succeeds whenever c does can be replaced by `trans(`s`, `α`, `t`, `(c, c')`)`. The uniqueness condition on the transition from s' is needed to preserve the branching behavior of the transition system. However, if transitions from s are all mutually exclusive (as in the case of conditional branches), the above rule can still be applied without violating the observable behavior of the system. Note that it is impossible to derive such optimizations solely from user annotations (such as *atomic* in SPIN) since the transitions that are merged span block boundaries.

This optimization can be implemented by suitably merging the rule sets when new locations are generated due to sequential composition (equation 4) and conditionals (equation 6).

Live Variable Optimization: Since each control state is associated with a set of variables, it is imperative that we keep only the "needed" variables in each state. We compute an upper approximation of the needed set as follows. Consider the creation of a new control state *mid* as an intermediate state in a sequential composition (equation 4). The variables in the expression E_1 as well as the variables in the control state *entry* are used before reaching *mid*. Similarly, the variables in E_2 and *exit* are potentially used after leaving *mid*. The definition of *vars*(*mid*) is the intersection of these two sets, and forms an upper approximation of the needed variable set.

Removing dead variables, i.e., those that are not needed, from state vectors not only lowers memory requirements for the state space search, but may reduce the size of the state space itself. Values of dead variables may increase the number of states in the system by discriminating between two instances of what is essentially one state. This fragmentation of states may be avoided by setting the dead variables to a default value manually (as recommended in the Murφ manual). It is encouraging to note that such workarounds can be effectively replaced by a simple program analysis.

State Representation: A control state in a concurrent automaton corresponds to the collection of control states of each sequential component, stored in some data structure. Transition rules are based on matching control states of individual components. Note that the control states of a concurrent automaton are maintained as a tree (using the symbol '$\times$' at the internal nodes) with the control states of component sequential automata at the leaves. Using this scheme, the control state of a sequential automaton can be accessed with no regard to changes that may occur (due to dynamic process creation) elsewhere in the concurrent automaton. However, when the structure of the concurrent process is known at compile time, the tree structure can be collapsed into a tuple, yielding the so called *state vector* representation. Moreover, the state vector representation enables a more compact representation of the concurrent state by squeezing together different components of the vector into a single memory word. Currently the compiler generates transition rules with state vector representation whenever possible, but without compression.

Partial Evaluation: Note that we have thus far not inspected the internals of the computation attached to each transition rule. Using simple partial evaluation [JGS93], we can eliminate from the rule sets the transitions whose computations are known to fail, and eliminate from the transition rules the computations that are known to succeed. Neither transformation affects the size of the explored state space. The former reduces the size of rule sets, while the latter optimizes application of rules.

4 Experimental Results

Consistent with the spirit of the XMC system, the XL compiler was implemented starting with an encoding of the equations in Figure 4 as a tabled logic program. The program was subsequently modified to implement the optimizations described in the previous section. The transition rules generated by the compiler are used to compute the global transitions as and when required by the XMC model checker.

In the following, we evaluate the effectiveness of the XL compiler, and compare the performance of the XMC system with and without compilation with that of Murφ [Dil96]. Performance measurements for Murφ were taken using version 2.70, since Murφ versions 3.0 and above do not support verification of liveness formulas.

The Benchmarks: We evaluate the performance of the XL compiler using the following four examples:

`i-Protocol` is a sliding window protocol in the GNU UUCP stack. The process structure is simple (a sender and a receiver processes connected via a pair of channel processes that can drop or corrupt data), although each sequential process itself is complex. Four versions of the i-Protocol were checked for presence of a livelock: for sliding window sizes (WS) of 1 and 2, and for each window size, one version with a bug that leads to a livelock and one fixed version.

`Rether` [CV95] is an Ethernet protocol that supports real-time traffic. Compared to the i-Protocol, the communication patterns for rether are more complex (any process can communicate with any other, depending on the global state), while each sequential process itself is relatively simple. The protocol was tested for a configuration of 5 processor nodes, with a maximum of 3 slots for real-time traffic in each Ethernet frame. Two properties, deadlock-freedom and starvation-freedom, were verified.

`Leader` is an encoding of the leader election protocol [DKR82] adapted from the SPIN example suite. The protocol was verified for 3, 5 and 7 participating nodes, communicating via buffered channels. The property verified was that exactly one leader is elected in any run of the protocol.

`Sieve` is an encoding of the sieve of Eratosthenes, also from the SPIN example suite. The program was tested for various number of filters in the sieve (3, 5 and 7), and the property verified corresponds to a correctness condition: that the sequence produced by the final filter has a specific value at a given position.

The last two examples were chosen mainly since they were originally used to evaluate the performance of XMC system [RRR+97] and hence form an useful point of reference. All benchmark instances except the buggy versions of i-Protocol require exploration of the entire reachable state space. This was chosen so that we can compare the performance of XMC's local model checker with that of Murφ's global checker.

Table 1 lists the explored state space of the various benchmarks in the two systems. The example specifications (XL as well as Murφ encodings), runtime parameters under which the following performance figures were obtained, as well as all experimental data are available from `http://www.cs.sunysb.edu/~lmc/compiler`.

Performance Measurement and Evaluation Criteria: Each of the three systems were evaluated on the four examples above in terms of verification time and space used during verification. All measurements were made on a Sun Enterprise 4000 with 2 GB main memory running Solaris 5.2.6. The times measured were CPU times reported by the different systems. For Murφ, verification time does not include time to generate C++ code from Murφ specifications and to compile it using g++. We used g++ (v2.8.1) with `-O4` option to obtain an optimized executable. The verification times for XMC were the CPU time needed to answer the corresponding `models` query. For Murφ it is the time to do reachability and assertion checks (followed by cycle detection for liveness formulae only). The verification times for Murφ were obtained *without* using symmetry reduction.

Benchmark	# of States Explored		# of Transitions	
	XMC	Murφ	XMC	Murφ
i-Protocol, WS=1, Bug	230	18672	319	93480
i-Protocol, WS=1, Fixed	14014	39280	51702	192772
i-Protocol, WS=2, Bug	1562	348580	2527	1733816
i-Protocol, WS=2, Fixed	134360	636004	491872	3121912
rether	336	2241	366	2801
sieve(3)	615	541	1423	1232
sieve(5)	4023	3367	12091	9830
sieve(7)	22941	19006	81703	65902
leader(3)	67	88	124	170
leader(5)	864	1456	2687	4678
leader(7)	11939	25632	52300	115594

Table 1: Characteristics of the benchmarks

Space usage measurements for XMC were obtained by adding the maximum space used in each of the memory areas of XSB: the table space, the four Prolog stacks, as well as permanent space (where dynamic code is kept). Space measurements for Murφ are the sizes of the state hash table reported by its statistics.

Analysis of Experimental Data: Table 2 lists the time and space performance of XMC with and without compilation, and Murφ on the examples described above. Observe from the table that compilation speeds up XMC by up to a factor of 15 for sieve, factors of 5 or better for the i-Protocol and around a factor of 4 for leader. Fixed version of the i-Protocol with window size 2 could not be verified using the original XMC system due to memory overflow, whereas it completes in under 4 minutes with compilation. On the other hand, the compiled code actually performs slightly worse than interpretation for rether. This is due to the large number of transition rules generated for rether arising from the ability of any two processes to communicate with one another. Many rules do not specify transitions from reachable states, and the presence of these rules imposes severe indexing overheads when selecting the applicable rule.

Note that all times for XMC with compilation are comparable to verification using Murφ. It should be noted that Murφ performs global checking and hence inspects all reachable states while XMC inspects only a portion of the state space that contributes to the proof/disproof of the given property. This accounts for the difference in performance for the buggy versions of i-Protocol.

The compilation time for XMC (not shown in the table) ranges from 0.1s to 0.2s, and is typically much smaller than the verification time. This compilation time includes the time to preprocess and load XL specifications, translate them to rules and load the rules. For Murφ, the time needed to generate the executable ranges from 7s for leader to 11s for the i-Protocol.

Observe that compilation also reduces memory requirements, by factors ranging from 2 to 15. The main reason is that the transition relation, which needed to be tabled (or cached) when computed by the interpreter, is precomputed by the compiler into a set of Prolog facts that do not need further tabulation. The savings in rether again appear minimal, mainly since the number of transition rules generated consumes a large portion of the permanent space in XSB, and dominates the memory measurements.

Benchmark	Time (sec)			Space (MB)		
	XMC (original)	XMC (compiled)	Murφ	XMC (original)	XMC (compiled)	Murφ
i-Protocol, WS=1, Bug	0.98	0.05	1.76	6.18	0.52	0.30
i-Protocol, WS=1, Fixed	99.72	12.82	7.33	388.85	18.31	1.61
i-Protocol, WS=2, Bug	1.30	0.31	35.61	13.01	0.78	5.58
i-Protocol, WS=2, Fixed	mem o/f	214.36	139.83	mem o/f	198.06	26.71
rether, deadlock free	0.19	0.22	0.20	0.78	0.58	0.03
rether, no starvation	0.38	0.47	0.49	0.87	0.64	0.10
sieve(3)	1.54	0.19	0.14	6.07	1.07	0.02
sieve(5)	15.57	1.20	0.92	55.07	7.42	0.17
sieve(7)	130.12	8.71	8.36	437.88	61.32	1.03
leader(3)	0.13	0.04	0.03	0.85	0.47	0.01
leader(5)	2.82	0.59	0.39	13.27	2.44	0.06
leader(7)	68.66	12.90	12.63	294.47	44.56	1.87

Table 2: Comparative performance of XMC (with and without compilation) and Murφ

The performance of XMC relative to SPIN is not shown in the table. For both buggy versions of the i-Protocol, XMC with compilation and SPIN (v3.2.3) show very similar performance in terms of time (0.05s and 0.31s for XMC vs. 0.04s and 0.4s for SPIN, for WS=1 and 2 respectively) as well as space (0.52M and 0.78M vs. 1.0M and 2.3M respectively). For fixed version with window size 1, SPIN takes 495s and consumes 1.1GB of memory; for window size 2, we have been unable to obtain SPIN numbers without using bitstate hashing or hash compaction, both of which compute only approximate answers. Although partial order reduction [HP95] does not change the verification performance for i-Protocol, it substantially reduces the search space for leader, making SPIN significantly outperform XMC, even with compilation. Currently, XMC does not perform partial order reduction; integrating such search-space reduction techniques into XMC is a topic of future work.

Finally, note that the space usage for Murφ is significantly lower than XMC with or without compilation. State vectors are stored in compressed form in Murφ. For instance, a state in i-Protocol (window size 2) is represented in 56 bits (without hash compaction). Recall that although the XL compiler encodes global states as state vectors, it does not further compress this representation. Study of compile-time techniques to reduce the memory needed to store the state space is a topic of current research.

Effectiveness of the Optimizations: We now present experimental results on the effectiveness of the individual optimizations described in Section 3.2.

The most significant of the optimizations is the elimination of `i`-transitions, since it can lead to considerable reductions in the state space. Eliminating `i`-transitions across block boundaries is particularly interesting. As mentioned in Section 3.2, the effect of these eliminations cannot be achieved by user annotations (such as *atomic* and *d_step* in SPIN) since the optimization merges transitions across block boundaries. To quantify the impact of this elimination, we turned off elimination of `i`-transitions whenever the candidate transitions spanned basic blocks (but performed elimination in all other cases) and measured the performance of XMC with compilation. Table 3 shows the time, memory, and state space figures, as well as the degradation factor relative to the those of XMC with compilation (which were given

Benchamrk	Time (sec)		Space (MB)		# of states		# of transitions	
i-proto, WS=1, Bug	1.72	(34x)	1.98	(4x)	9125	(40x)	18180	(57x)
i-proto, WS=1, Fixed	300.28	(23x)	223.98	(12x)	188112	(13x)	664528	(13x)
i-proto, WS=2, Bug	2.41	(8x)	2.57	(3x)	12483	(8x)	23729	(9x)
i-proto, WS=2, Fixed	out of memory							
rether, deadlock free	0.21	(1x)	0.59	(1x)	341	(1x)	371	(1x)
rether, no starvation	0.49	(1x)	0.66	(1x)	341	(1x)	371	(1x)
sieve(3)	2.11	(11x)	10.22	(10x)	8323	(14x)	28793	(20x)
sieve(5)	47.79	(40x)	45.15	(6x)	123147	(31x)	573909	(47x)
sieve(7)	out of memory							
leader(3)	0.03	(1x)	0.53	(1x)	88	(1.3x)	170	(1.4x)
leader(5)	0.72	(1.2x)	3.42	(1.4x)	1456	(1.7x)	4678	(1.7x)
leader(7)	21.54	(1.7x)	81.34	(1.8x)	25632	(2.1x)	115594	(2.2x)

Table 3: Effect of *not* eliminating `i`-transitions across block boundaries

in Table 2). The tables reveal that the elimination of `i`-transitions across block boundaries reduces state spaces by more than a factor of 12 for i-Protocol, by 2 for leader and 30 for sieve. Again, the savings in rether appear minimal due to the simplicity of the sequential components of the protocol. The state space reduction also translates directly to savings in verification time and space, as the table shows.

As noted in Section 3.2, eliminating dead variables from state vectors can lower the memory requirements to store the state space, and sometimes reduce the size of the state space itself. With dead variable elimination, we observed a 7% drop in memory requirements across all examples. For i-Protocol, the elimination lowered the size of the state space by 5%, but there was no change in state space for the other examples.

Partial evaluation of expressions does not affect the state space, but can reduce the number of transition rules generated and eliminate some computation from the rules. For the examples described above, this optimization reduced the set of the generated rules by 25% on the average, and changed the verification time and space by less than 5%, mainly from reduction of indexing overheads.

5 Discussion

We have shown that compiling high-level specifications into global transition rules improves model checking performance. We also showed that such compilation can be performed very efficiently. Although presented as a compilation technique for XL, the technique can be readily adapted to translate high-level specifications written in other formalisms also. We showed that combining computations across block boundaries, an optimization that cannot be done based on user annotations alone, reduces state space significantly. This optimization can be introduced to improve the performance of any explicit-state model checker. Although dead variable elimination does not lead to the kind of performance improvements shown by the above optimization, it is nevertheless useful especially in a language with imperative constructs such as loops and assignments. Partial evaluation can be used to reduce the indexing overheads: those associated with finding transitions in the global automata from a given state. Deriving a generic optimizing compiler, based on systems such as PAC [CMS95], will help share such optimizations over a wide variety of specification languages and verification systems.

While we have offered preliminary evidence of the importance of optimizing compilation, its full power remains to be exploited. For instance, can powerful state-space reduction techniques such as partial order reduction [HP95, KLM+98] be used at the transition rule level to eliminate entire families of transitions in one step, at compile time? What techniques and optimizations are most useful for reducing the space requirements for verification? We believe that answers to these questions will lead to considerable improvements in the efficiency of current model checkers.

Acknowledgements: We thank Gerard Holzman for his invaluable help in collecting accurate figures on SPIN's performance on the i-Protocol, and Xiaoqun Du for encoding and evaluating the verification of i-Protocol on SPIN and original XMC. We also thank Rance Cleaveland, I.V. Ramakrishnan, Scott Smolka and David Warren for fruitful discussions.

References

[App92] A.W. Appel. *Compiling with Continuations.* Cambrisge University Press, 1992.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the iso specification language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Dias, editors, *The Formal Description Technique LOTOS*, pages 23–76. North-Holland, 1989.

[CDD+98] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In *Programming Languages, Implementations, Logics and Programming (PLILP'98).* Springer Verlag, 1998.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CMS95] R. Cleaveland, E. Madelaine, and S. Sims. Generating front ends for verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS '95).* Springer Verlag, 1995.

[CV95] T. Chiueh and C. Venkatramani. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCPMM'95*, pages 27–37, 1995.

[CW96a] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[CW96b] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[DDR+99] Y. Dong, X. Du, Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, O. Sokolsky, E.W. Stark, and D.S. Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS '99)*. Springer Verlag, 1999.

[Dil96] D. L. Dill. The Murφ verification system. In *Computer Aided Verification (CAV'96)*, pages 390–393. Springer Verlag, 1996.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[GS90] H. Garavel and J. Sifakis. Compilation and verification of lotos specifications. In *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394, 1990.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP95] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Seventh Int. Conf. on Formal Description Techniques (FORTE '94)*, pages 177–194. Chapman and Hall, 1995.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[KLM+98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, pages 345–357. Springer Verlag, 1998.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Verlag, 1995.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Computer-Aided Verification (CAV '97)*. Springer Verlag, 1997.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[XSB98] XSB. The XSB logic programming system, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog/XSB/`.