AN OPTIMIZING COMPILER FOR EFFICIENT MODEL CHECKING

Yifei Dong, C.R. Ramakrishnan
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
ydong@cs.sunysb.edu, cram@cs.sunysb.edu

Abstract We present an optimizing compiler that translates high-level specifications in a language based on value-passing CCS into *rules* representing the global transition relation of the underlying automata. The transitions can be computed from these rules in unit time (modulo indexing) during verification. The compiler takes time that is, in the worst case, quadratic in the number of parallel components of the system. The compiler incorporates several optimizations to reduce the state space of the generated automata. We present experimental results which show that our technique reduces the memory and time needed for verification significantly: by more than an order of magnitude in some cases.

Keywords: Model Checking, Specification Languages, Optimization.

1. INTRODUCTION

Specification languages supported by most verification tools have high-level features such as data structures, procedures, and parameterized processes. Model checking techniques [5, 18, 6] typically view the specification in terms of the underlying automaton: labeled transition system or Kripke structure [16]. Verification tools first *compile* the input specifications into a compact automata representation. For example, SPIN [11] compiles PROMELA specifications into local automata, one for each process; CÆSAR [10] translates LOTOS specifications [2] into Petri nets. At verification time, these representations are used by the explicit-state model checkers to construct the global automaton.

J. Wu et al. (eds.), *Formal Methods for Protocol Engineering and Distributed Systems*

In contrast to the above tools, Murφ [8] provides a guarded command language for specifying a system directly as rules defining its global transition relation. This permits the user to 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 shows that the low-level specification enables Murφ to be considerably faster, and consume much less memory, than other tools [9].

It would hence be ideal to combine the verification efficiency afforded by low-level specifications with the ease of correct encoding enabled by high-level specifications. Many tools support user annotations 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 user must ensure that the optimizations are correct. Moreover, 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 efficiently translates high-level specifications (based on value-passing CCS [17]) into a compact representation of the global automaton. We use several optimizations to 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 global transition relation. It precomputes possible synchronizations at compile-time while still maintaining polynomial (quadratic) bounds for compile time and space. Transitions of the global automaton can be computed from these rules in unit time (modulo indexing) during verification.

- The compiler is derived directly from the structural operational semantics of value-passing CCS expressions. In contrast, the translation for LOTOS programs described in [10] is not directly based on LOTOS operational semantics [2].

- Consecutive computation steps are grouped into single atomic transitions *even across basic block boundaries.* In contrast *atomic* annotations in SPIN can group together only computations that lie within a single block. Our aggressive optimization reduces the state space by more than an order of magnitude in some examples, thereby improving verification performance.

Our Approach and its Effectiveness. We describe the compilation technique using the XMC system [19], a model checker for modal mu-calculus [13] and XL, a process description language. XL, described formally in Section 2, 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 [22]. The core of the system consists of two predicates, `trans`:*State*× *Action* × *State* which computes the transition relation by interpreting XL process terms, and `models`:*State* × *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. This encoding reduces model checking to logic program query evaluation. Furthermore, the goal-directed query evaluation mechanism of the XSB logic programming system yields a local model checker. 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 [7].

The compiler for XL is described in Section 3. We implemented the compiler in the XMC system and evaluated its effectiveness using benchmark programs with different characteristics. The results of the experiments are presented in Section 4. 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. In Section 5, we describe the applicability of our technique to other tools.

2. THE SPECIFICATION LANGUAGE XL

XL is based on the value passing CCS [17]. Values and computations are represented as Prolog terms and predicates respectively. Thus the specifications can use recursive data structures and computations.

The syntax of XL specifications are described by the grammar shown in Figure 1.1. 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 that matches term t over port `chan`; `out(chan(`t`))` outputs the value repre-

E	$\longrightarrow$	*Comp*	(computation)
	|	`in(`*Term*`)`	(input communication)
	|	`out(`*Term*`)`	(output communication)
	|	`zero`	(unit deadlocked process)
	|	E `o` E	(sequential composition)
	|	E `#` E	(choice)
	|	`if(`*Comp*`,` E`,` E`)`	(conditional)
	|	E `|` E	(parallel composition)
	|	E `\` *PortSet*	(restriction)
	|	E `@` *PortMap*	(relabeling)
	|	*Proc*	(process invocation)
Def	$\longrightarrow$	(*Proc* `::=` E)*	(process definitions)

Figure 1.1 Syntax of XL

sented 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 language features are illustrated by the complete specification of the Alternating Bit Protocol [21] in Figure 1.2.

Operational Semantics of XL. In the following, we assume familiarity with the notions of terms and substitutions [15]. A substitution θ is a *unifier* of two terms t_1 and t_2 if $t_1\theta = t_2\theta$. 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 1.3 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

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

send(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 send(NSeq)     % successful ack, next message
              , send(Seq))                     % unexpected ack, resend message
        ) # send(Seq)).                        % upon timeout, resend message

recv(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 recv(NSeq) )
        , out(ackOut(RecSeq)) o recv(Seq)).   % unexpected seq, resend ack

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

Figure 1.2 Specification of the Alternating Bit Protocol in XL

transmitted by synchronizing $\mathtt{in}(t)$ with $\mathtt{out}(t')$ 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 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 a list L of terms, $s \in L$ if there is a term t in L such that $s = \mathtt{in}(t)$ or $s = \mathtt{out}(t)$. Similarly, rule 14 uses a relabeling function as follows: given a list F of pairs of terms, if s/t is a member of F then $F(\mathtt{in}(t)) = \mathtt{in}(s)$ and $F(\mathtt{out}(t)) = \mathtt{out}(s)$; if there is no s such that s/t is a member of F then $F(t) = t$.

3. COMPILING XL

Given an XL program, the compiler derives a set of rules that concisely describes the corresponding transition relation. The conciseness

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

$$(3)\ \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \circ F,\theta \xrightarrow{\alpha} E' \circ F,\theta'} \qquad (4)\ \frac{[\![Comp]\!]_L\theta = \theta' \ \wedge\ E,\theta' \xrightarrow{\alpha} E',\theta''}{Comp \circ E,\theta \xrightarrow{\alpha} E',\theta''}$$

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

$$(7)\ \frac{[\![C]\!]_L\theta \neq \{\} \ \wedge\ E_1,\theta \xrightarrow{\alpha} E_1',\theta'}{\texttt{if}(C,\ E_1,\ E_2),\theta \xrightarrow{\alpha} E_1',\theta'} \qquad (8)\ \frac{[\![\texttt{not}(C)]\!]_L\theta \neq \{\} \ \wedge\ E_2,\theta \xrightarrow{\alpha} E_2',\theta'}{\texttt{if}(C,\ E_1,\ E_2),\theta \xrightarrow{\alpha} E_2',\theta'}$$

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

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

$$(12)\ \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{P',\sigma\theta \xrightarrow{\alpha} E',\sigma\theta'} \quad (P ::= E, \sigma = mgu(P,P'))$$

$$(13)\ \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \setminus L,\theta \xrightarrow{\alpha} E' \setminus L,\theta'} \quad (\alpha \notin L) \qquad (14)\ \frac{E,\theta \xrightarrow{\alpha} E',\theta'}{E \,@\, F,\theta \xrightarrow{F(\alpha)} E' \,@\, F,\theta'}$$

Figure 1.3 Operational Semantics of XL

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 $\pi\sigma$, i.e. a control state π under a substitution σ. Transitions between states $\pi_s\sigma_s$ and $\pi_d\sigma_d$ can be captured by a transition rule of the form $\texttt{trans}(\pi_s,\ \alpha,\ \pi_d,\ c)$, where α is the label and c relates σ_s and σ_d. Formally, given a set of transition rules $\mathcal{T}$, the underlying transition relation is computed using the inference rule:

$$\frac{\texttt{trans}(\pi_s,\ \alpha,\ \pi_d,\ c) \in \mathcal{T} \ \wedge\ [\![c]\!]_L\theta_s = \theta_d}{\pi_s\theta_s \xrightarrow{\alpha} \pi_d\theta_d} \tag{1.1}$$

We compile each process expression into an automaton with two distinguished control states: *entry* and *exit*. The compiler is described in

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

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

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

$[\![E_1 \circ E_2]\!]\ entry\ exit = [\![E_1]\!]\ entry\ mid \cup [\![E_2]\!]\ mid\ exit$ ··· (4)

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

$[\![E_1 \ \# \ E_2]\!]\ entry\ exit = [\![E_1]\!]\ entry\ exit \cup [\![E_2]\!]\ entry\ exit$ ··· (5)

$[\![\texttt{if}(C, E_1, E_2)]\!]\ entry\ exit$ ··· (6)

$= \{\ \texttt{trans}(entry,\ \texttt{i},\ iftrue,\ C),$
$\quad \texttt{trans}(entry,\ \texttt{i},\ iffalse,\ \texttt{not}(C))\ \}$
$\cup\ [\![E_1]\!]\ iftrue\ exit$
$\cup\ [\![E_2]\!]\ iffalse\ exit$

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

$[\![E_1 \mid E_2]\!]\ entry\ exit = \{\ \texttt{trans}(s_1 \times V_2,\ \alpha_1,\ d_1 \times V_2,\ c_1)$: ··· (7a)
$\quad \texttt{trans}(s_1, \alpha_1, d_1, c_1) \in [\![E_1]\!]\ entry\ exit$
$\quad$ and V_2 is a fresh variable$\}$

$\cup\ \{\ \texttt{trans}(V_1 \times s_2,\ \alpha_2,\ V_1 \times d_2,\ c_2)$: ··· (7b)
$\quad \texttt{trans}(s_2, \alpha_2, d_2, c_2) \in [\![E_2]\!]\ entry\ exit$
$\quad$ and V_1 is a fresh variable$\}$

$\cup\ \{\ \texttt{trans}((s_1 \times s_2)\theta,\ \texttt{tau},\ (d_1 \times d_2)\theta,\ (c_1, c_2)\theta)$: ··· (7c)
$\quad \texttt{trans}(s_1, \alpha_1, d_1, c_1) \in [\![E_1]\!]\ entry\ exit$
$\quad \wedge\ \texttt{trans}(s_2, \alpha_2, d_2, c_2) \in [\![E_2]\!]\ entry\ exit$
$\quad$ and θ is the most general substitution such that
$\quad \{\alpha_1, \alpha_2\} \equiv \{\texttt{in}(t),\ \texttt{out}(t\theta)\}$ for some term t $\}$.

$[\![E \backslash L]\!]\ entry\ exit = \{\ \texttt{trans}(\rho_{E,L}(s),\ \alpha,\ \rho_{E,L}(d),\ c)$: ··· (8)
$\quad \texttt{trans}(s, \alpha, d, c) \in [\![E]\!]\ entry\ exit \ \wedge\ \alpha \notin L\ \}$

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

$[\![E @ F]\!]\ entry\ exit = \{\ \texttt{trans}(\rho_{E,F}(s),\ F(\alpha),\ \rho_{E,F}(d),\ c)$: ··· (9)
$\quad \texttt{trans}(s, \alpha, d, c) \in [\![E]\!]\ entry\ exit\ \}$

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

$[\![Proc]\!]\ entry\ exit = \{\ \texttt{trans}(entry,\ \texttt{i},$ ··· (10a)
$\quad ep(Proc)(\langle exit \rangle \oplus subterms(Proc)),\ \texttt{true})\ \}$
$\cup\ [\![definitions(Proc)]\!]$ ··· (10b)

where $ep(Proc)$ 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'.

$[\![Proc ::= E]\!] = [\![E]\!]\ ep(Proc)(\langle Cont \rangle \oplus subterms(Proc))\ \ Cont$ ··· (11)

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

Figure 1.4 Compilation Rules for XL

Figure 1.4 in terms a semantic function $[\![E]\!]\ entry\ exit$ which maps process expression E to a set of the transition rules. In the figure, we use $\langle \ldots \rangle$ to denote sequences and $\oplus$ to denote concatenation of sequences.

The compiler views computation and process invocation in XL as state-changing steps that are not observable. Such steps are represented by transitions with a special label, 'i', denoting internal transitions (see equations 1, 6, and 10). Since i-transitions are unobservable, we can define a form of equivalence called i-equivalence. Informally, two processes P and Q are i-equivalent if P and Q have identical branching behaviors when i-transitions are ignored. Under i-equivalence the correctness of the compiler w.r.t. XL semantics can be readily shown.

The first six equations in Figure 1.4 directly encode the semantic rules 1–8 in Figure 1.3. The remaining equations are explained below.

Compiling Process Invocations. 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 [1] naturally limits the number of i-transitions introduced by the compiler. Parameter 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.

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.

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 precomputes synchronizations *without* incurring an exponential blow up of transition rules, as explained below. 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, the number of transition rules generated is usually much smaller than the above bound. For instance, the number of transition rules for the leader election protocol as well as the sieve of Eratosthenes grow linearly with the number of processes.

The compiler is implemented in the XSB tabled logic programming system [22] by simply encoding the set equations in Figure 1.4 as a Horn clause program and evaluating it using tabled resolution [3]. Termination of the 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. The scheme can be extended to compile parameterized processes (such as `p(N)` above) whenever the value of the parameter is known at compile time. However, extending the scheme to *dynamically* compile processes that are created at verification time remains an interesting open problem.

Reducing Internal (i-) Transitions. Since i-transitions are not observable, we can reduce the system state space modulo i-equivalence, i.e., replacing sequences of transitions of the form $\mathtt{i}^*\alpha\mathtt{i}^*$ with α whenever the system's branching behavior is not changed. We call such an optimization as i-elimination. Note that i-transitions are only generated by sequential statements, namely computation, conditional, and process invocation. Therefore it is sufficient to describe i-elimination only for sequential components. It should be noted that state space reduction in sequential components translates into corresponding (multiplicative) reduction in the state space of a parallel processes.

In our formulation of i-elimination, we need to distinguish between i-transition rules that arise from computations (equations 1 and 6) and those that arise from process invocation (equation 10). We use labels $\mathtt{i}_c$ and $\mathtt{i}_p$ to denote these two kinds of transitions respectively whenever such a distinction is necessary. The i-elimination optimization can be defined as a sequence of rewrites on the set of transition rules $\mathcal{T}$ using the following rules:

Forward elimination: Let $I = \texttt{trans}(s, \texttt{i}_c, t, c) \in \mathcal{T}$. Then,
$\mathcal{T} := (\mathcal{T} - \{I\}) \cup \{ \texttt{trans}(s, \alpha, t'\theta, (c \wedge c'\theta)) \mid \texttt{trans}(s', \alpha, t', c') \in \mathcal{T}$ and there exists a substitution θ such that $t = s'\theta\}$.

Backward elimination: Let $I = \texttt{trans}(s, \texttt{i}, t, c) \in \mathcal{T}$ such that $\forall\, \texttt{trans}(s, \alpha'', t'', c'') \in \mathcal{T} - \{I\}$, the computations c and c'' are mutually exclusive, i.e., $\forall$ substitutions σ $[\![c]\!]_L\sigma \cap [\![c'']\!]_L\sigma = \{\}$. Let $R = \texttt{trans}(s', \alpha, t', c') \in \mathcal{T}$ such that $\alpha \neq \texttt{i}_p$, and θ be a substitution such that $t'\theta = s$. Then,
$\mathcal{T} := (\mathcal{T} - \{I, R\}) \cup \{\texttt{trans}(s', \alpha, t, (c' \wedge c\theta), \texttt{trans}(s', \alpha, t', (c' \wedge \texttt{not}(c)\theta)\}$

Eliminating $\texttt{i}_p$-transitions by forward elimination, which is currently not permitted, will be analogous to inlining. This is explicitly avoided since additional conditions are needed to ensure that the inlining terminates even in the presence of recursion in processes. Also note that forward elimination may exponentially increase the size of rule sets since shared computations may be repeated on independent transitions; however, the benefits of the optimization appear to outweigh the potential (although rare) blow ups. For backward elimination, the mutual exclusion condition on I is needed to preserve the system's branching behavior.

Each application of an i-elimination optimization can be shown correct with respect to i-equivalence. We can also show that any sequence of i-eliminations will terminate. Although presented as an off-line optimization, both elimination steps can be performed while applying the compilation rules by suitably merging the rule sets whenever new control states are generated.

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 increase may be avoided by setting the dead variables to a default value manually,

```
trans(abp(A,chan_1(B,C),D,E), out(drop), abp(A,chan_0(C),D,E), true).
trans(abp(A,B,chan_1(C,D),E), out(drop), abp(A,B,chan_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(send_0(A,B),chan_0(C),D,E), tau,
      abp(send_1(A,B),chan_1(A,C),D,E), true).
trans(abp(send_1(A,B),C,chan_1(D,E),F), tau,
      abp(send_0(G,B),C,chan_0(E),F), (D == A,  G is 1 - A)).
trans(abp(send_1(A,B),C,chan_1(D,E),F), tau,
      abp(send_0(A,B),C,chan_0(E),F), (not(D == A))).
trans(abp(A,chan_1(B,C),D,receiver_0(E,F)), tau,
      abp(A,chan_0(C),D,receiver_4(B,G,F)), (B == E,  G is 1 - E)).
trans(abp(A,chan_1(B,C),D,receiver_0(E,F)), tau,
      abp(A,chan_0(C),D,receiver_3(B,E,F)), (not(B == E))).
trans(abp(A,B,chan_0(C),receiver_4(D,E,F)), tau,
      abp(A,B,chan_1(D,C),receiver_0(E,F)), true).
trans(abp(A,B,chan_0(C),receiver_3(D,E,F)), tau,
      abp(A,B,chan_1(D,C),receiver_0(E,F)), true).
```

Figure 1.5 Transition Rules for the Alternating Bit Protocol

and is a fairly common encoding trick adopted by experienced users. It is encouraging that such workarounds can be effectively replaced by a simple program analysis (see Section 4 for details).

Finally, note that the control states of the global transition relation are maintained as a tree with '×' as internal nodes and the control states of the component sequential processes as the leaves. When the structure of the concurrent process is known at compile time, the tree structure can be collapsed into a tuple, yielding the *state vector* representation. Currently the compiler generates rules in terms of state vectors whenever possible, but without compressing components of the vector itself.

In Figure 1.5, the transition rules generated by the compiler for the alternating bit protocol (Figure 1.2) are shown as an illustration.

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 1.4 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, and the performance of Murφ [8]. 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. The following examples were used as benchmarks:

`i-Protocol` (*i-p* in the tables) is a sliding window protocol in the GNU UUCP stack. The sender and receiver processes have complex internal (sequential) structure. Presence of a livelock was checked for sliding window sizes (WS) of 1 and 2, each with and without the livelock.

`Rether` [4] is an Ethernet protocol that supports real-time traffic. Compared to `i-Protocol`, each node of `rether` is a relatively simple process, while the communication patterns are more complex (each node can communicate with any other). The protocol was tested with 5 nodes and up to 3 real-time slots per Ethernet frame. Two properties, deadlock-freedom (*dlf*) and starvation-freedom (*ns*), were verified.

`Leader` and `sieve` are taken from the SPIN example suite. For `leader`, the property verified was that exactly one leader is elected in any run. For `sieve`, we verified that the sequence produced by the final filter has a specific value at a given position. The example specifications, runtime parameters used, and experimental data are available from `http://www.cs.sunysb.edu/~lmc/compiler`.

Performance Measurement and Evaluation Criteria. 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 is the CPU time used by the executable obtained by compiling the generated C++ code using g++ (v2.8.1) (with `-O4` for max. optimization). The times for Murφ were obtained *without* using symmetry reduction. Verification times for XMC is the CPU time needed to answer the corresponding `models` query. Space usage for XMC was measured as the sum of the maximum space used in each of the memory areas of XSB: the table space, the four Prolog stacks, as well as permanent space (which includes space for dynamic code). Space usage for Murφ was taken as the sizes of the state hash table reported by its statistics.

Analysis of Experimental Data. Table 1.1 lists the time and space performance of XMC (with and without compilation) and Murφ on the benchmark examples. Observe from the table that compilation speeds up XMC by up to a factor of 15 for `sieve`, factors of 5 or better for `i-Protocol` and around a factor of 4 for `leader`. On the other hand, the compiler slows the verification of `rether`. This is due to the severe indexing overheads added by the presence of "dead" rules: those that specify transitions from states that are unreachable at verification time.

Note that all times for XMC with compilation are comparable to verification using Murφ. XMC's local model checker explores only a subset

Benchmark	*Time (sec)*			*Space (MB)*		
	XMC	*XMC/c*	*Murφ*	*XMC*	*XMC/c*	*Murφ*
i-p(1, bug)	0.98	0.05	1.76	6.18	0.52	0.30
i-p(1, fixed)	99.72	12.82	7.33	388.85	18.31	1.61
i-p(2, bug)	1.30	0.31	35.61	13.01	0.78	5.58
i-p(2, fixed)	out of mem	214.36	139.83	out of mem	198.06	26.71
rether(dlf)	0.19	0.22	0.20	0.78	0.58	0.03
rether(ns)	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 1.1 Performance of XMC (with and without compilation) and Murφ

of reachable states for the buggy versions of `i-Protocol`, but explores all reachable states for other examples. In contrast, Murφ performs global checking, and hence explores all reachable states for all examples.

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 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 `i-Protocol`.

Observe that compilation also reduces memory requirements, by factors ranging from 2 to 15. In contrast to the original XMC system, the transition relation is precomputed by the compiler into a set of Prolog facts and need not be tabled (cached). Again, the savings in `rether` is minimal, since XSB's permanent space, where the transition rules are stored, dominates memory usage.

The performance of XMC relative to SPIN is not shown in the table. For both buggy versions of `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 bit-state hashing or hash compaction, both of which compute only approximate answers. Although partial order reduction [12] does not change the verification performance for `i-Protocol`,

Benchamrk	*Time (sec)*		*Space (MB)*		*# of states*		*# of trans.*	
i-p(1, Bug)	1.72	(34x)	1.98	(4x)	9K	(40x)	18K	(57x)
i-p(1, Fixed)	300	(23x)	224	(12x)	188K	(13x)	664K	(13x)
i-p(2, Bug)	2.41	(8x)	2.57	(3x)	12K	(8x)	23K	(9x)
i-p(2, Fixed)	out of memory							
rether(dlf)	0.21	(1x)	0.59	(1x)	341	(1x)	371	(1x)
rether(ns)	0.49	(1x)	0.66	(1x)	341	(1x)	371	(1x)
sieve(3)	2.11	(11x)	10.22	(10x)	8K	(14x)	28K	(20x)
sieve(5)	47.8	(40x)	45.15	(6x)	123K	(31x)	574K	(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)	5K	(1.7x)
leader(7)	21.5	(1.7x)	81.3	(1.8x)	26K	(2.1x)	116K	(2.2x)

Table 1.2 Effect of *not* eliminating i-transitions across block boundaries

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 compressed into bit-vectors in Murφ, thereby reducing space usage; for instance, a state in `i-Protocol` (WS=2) is stored using only 56 bits (without hash compaction). In contrast, the XL compiler generates uncompressed state vectors, leading to higher memory requirement.

Effectiveness of the Optimizations. Elimination of i-transitions leads to considerable reductions in the state space. The effect of performing this optimization across block boundaries is of special interest, since this cannot be done by user annotations (such as *atomic* and *d_step* in SPIN) alone. We measured this effect by turning off i-elimination whenever the candidate transitions spanned basic blocks, but performed elimination in all other cases. Table 1.2 shows the degradation in time and space performance, as well as relative state space sizes when the optimization is turned off. The savings due to this optimization are substantial in all examples except `rether`, again due to the simplicity of the sequential components of the protocol.

As noted in Section 3, eliminating dead variables from state representation can reduce the state space size, in addition to lowering the memory requirements. With dead variable elimination, we observed a 7% drop in memory requirements across all examples. For `i-Protocol`,

the elimination reduced the state space by 5%, but there was no change in state space for the other examples.

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. In fact, this optimization has been recently added to SPIN 3.3.0 and reportedly has led to significant performance gains (see [20]).

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 [12, 14] 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 considerably improve the performance of current model checkers.

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

This research was supported in part by grants CDA–9303181, EIA–9705998, and CCR–9711386 from the National Science Foundation.

References

[1] A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] 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.

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

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

[5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Workshop on Logic of Programs, LNCS 131*, pages 52–71, Yorktown Heights, 1981.

[6] 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.

[7] 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)*, 1998.

[8] D. L. Dill. The Murφ verification system. In *Computer Aided Verification (CAV'96)*, 1996.

[9] 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)*, 1999.

[10] H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In *Protocol Specification, Testing and Verification (PSTV '90)*, 1990.

[11] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

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

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

[14] 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)*, 1998.

[15] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

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

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

[18] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *International Symposium on Programming, LNCS 137*, Berlin, 1982.

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

[20] SPIN Newsletter, Number 25, May 1999. Available from `http://netlib.bell-labs.com/netlib/spin/news/news25.html`.

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

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