An Optimizing Compiler for Batches of Temporal Logic Formulas

James Ezick
Department of Computer Science
Cornell University
ezick@cs.cornell.edu

ABSTRACT

Model checking based on validating temporal logic formulas has proven practical and effective for numerous software engineering applications. As systems based on this approach have become more mainstream, a need has arisen to deal effectively with large batches of formulas over a common model. Presently, most systems validate formulas one at a time, with little or no interaction between validation of separate formulas. This is the case despite the fact that, for a wide range of applications, a certain level of redundancy between domain-related formulas can be anticipated.

This paper presents an optimizing compiler for batches of temporal logic formulas. A component of the Carnauba model checking system, this compiler addresses the need to handle batches of temporal logic formulas by leveraging the framework common to optimizing programming language compilers. Just as traditional optimizing compilers attempt to exploit redundancy and other solvable properties in a program to reduce the demand on a runtime system, this compiler exploits similar properties in groups of formulas to reduce the demand on a model checking engine. Optimizations are performed via a set of distinct, interchangeable optimization passes operating on a common intermediate representation. The intermediate representation captures the full modal mu-calculus, and the optimization techniques are applicable to any temporal logic subsumed by that logic. The compiler offers a unified framework for expressing some well understood single-formula optimizations as well as numerous inter-formula optimizations that capitalize on redundancy, logical implication, and, optionally, model-specific knowledge. It is capable of working either in place of, or as a preprocessor for, other optimization algorithms. The result is a system that, when applied to a potentially heterogeneous collection of formulas over a common problem domain, is able to measurably reduce the time and space requirements of the subsequent model checking engine.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Modal logic*

General Terms

Verification

Keywords

Optimizing compiler, Temporal logic, Model checking

1. INTRODUCTION

Model checking systems that validate temporal logic formulas with respect to a model are pervasive. These systems have numerous applications to software engineering problems including code comprehension, verification, and bug-finding. When presented with multiple formulas, most of these systems process them individually in a serial fashion. This is despite the fact that, for many problem domains, batches of temporal formula queries are generated that share some degree of commonality. For these systems, commonality can translate into redundant computation and storage that can be expensive when the model is large. This suggests a need for a preprocessing tool that can detect and utilize this commonality to reduce the workload handed off to the model checking engine.

This paper makes the following contributions. It explains how a compiler framework can be used to organize an optimization system for batches of temporal logic formulas. It demonstrates how some core optimizations for logic formulas can be partitioned and sequenced as independent optimization routines. It illustrates how an intermediate language can be used to unify optimization techniques for many practical logics. Finally, it presents experimental evidence that such a system reduces the complexity of batches of formulas useful for program analysis, and that this reduction translates into improved model checking performance that is orthogonal to any further improvement achieved by reducing the state space over which the formulas are later checked.

The modern optimizing programming language compiler provides an ideal paradigm for organizing a tool designed to manipulate batches of temporal logic formulas. Optimizing compilers work by translating a range of programming languages into a common intermediate language that is both the input and output format for an interchangeable collection of optimization routines. In this framework, each routine performs a specific optimization within the intermediate representation. Passes may interact by opening opportunities for others to make progress, but in general they do not exchange information beyond that which is included in the current incarnation of the program. Such an organization provides maximum flexibility with minimal overhead.

This paper presents an optimizing compiler for temporal logic formulas based on such a framework. The compiler can handle formulas in any logic for which a front end can be supplied to translate the logic into the modal mu-calculus [19]. Internally, the compiler manipulates mu-calculus formulas in equation block form [6, 7] and uses an extended version of this form as its intermediate representation. Optimization then occurs over this logic language.

The optimizer is organized into eight independent, dynamically scheduled optimization routines. Each routine performs a separate optimization designed to reduce the complexity of the equation block system as measured by the time and space required to subsequently resolve it. Many of these optimizations mirror traditional programming language optimizations. Each pass is capable of operating in either a model-dependent or model-independent mode. In the former, knowledge of the model is used to refine the notion of equivalence among modalities and atomic propositions, and thus give a more complete result. In the latter, these constructs are treated as abstract sets and only model-invariant equivalence and implication are detectable. The model-dependent mode can also incorporate some axiomatic knowledge of the successor relation into the optimization process. These distinctions are transparent to the individual passes. The result is a system in which generic batches of queries can be optimized before a specific model is supplied and then further refined afterwards.

The modal mu-calculus is an exponential-time decidable logic [11]. This has been demonstrated by the existence of both automata-theoretic [25] and automata-free [22] algorithms. Despite these results, the goal of the optimizer is not completeness. Rather, it is to provide a framework for a tractable set of optimizations over potentially very large query sets that take advantage of query boundaries and, optionally, model-specific knowledge. Some of the optimizations test for sufficient though not necessary conditions to perform certain reductions. These conditions can, in some cases, be replaced by the aforementioned decidability algorithms to provide a more complete, though significantly more time-intensive, optimization. Numerous other approaches to query optimization exist [15, 16, 17, 23]. Some perform single query optimizations beyond what is demonstrated in this paper. For applications for which these additional optimizations are desirable, this framework can be augmented with additional passes or, alternatively, could be used as a preprocessing tool for performing certain optimizations before passing to a system capable of performing these additional, domain-specific optimizations. This idea is discussed in more detail in the conclusion.

The compiler has been integrated into the Carnauba model checking system [14], a system for model checking aspects of C programs represented as unrestricted hierarchical state machines [2]. These are collections of ordinary state machines organized to reflect the call and return structure of imperative programs. Experiments with batches of queries of the sort used in software verification show that an optimizing compiler framework is both effective and efficient at reducing the total complexity of the workload passed to the model checking engine. Further, the collection of passes provided subsumes many logic-specific optimizations and reductions. In some cases, the optimized forms have no preimage in the native logic from which they originated. This indirectly eliminates the need for "extended operators" that are sometimes used to represent these forms.

The remainder of this paper is organized as follows: Section 2 discusses the details of the framework as well as introduces the syntax and semantics of the intermediate representation. In Section 3, the specific optimizations are enumerated and described. Section 4 presents empirical results from applying the compiler to batches of formulas generated from several benchmark programs. In Section 5, observations, conclusions, and directions for future work are discussed. Finally, Section 6 contains acknowledgments.

2. THE COMPILER FRAMEWORK

The framework of this compiler is designed to mirror that of a modern programming language compiler [1]. It has a well-defined intermediate language. A front end converts batches of temporal logic formulas into the intermediate language for optimization, while a back end converts code in the intermediate language into a form that can be fed into the Carnauba model checking engine. An external symbol table records peripheral information from the front end translation and feeds this information on demand to the optimizer. The optimizer, in turn, updates this information as transformations are performed. These components are described next. A full discussion of the routines and scheduling comprising the optimizer is left for Section 3.

2.1 Front End

The front end translates batches of temporal logic formulas, expressed as lists of queries, into the intermediate language. Each query consists of a minimum of two pieces of information: a formula and a tag denoting its source logic. The front end processes each query in turn, using the tag to select the appropriate routine for translating the formula into the modal mu-calculus, $L\mu$. Translations [9, 13] are known for many logics that have proven to be useful for model checking, including *CTL* [5], *FCTL* [12], CTL^* [10], and *PDL-Δ* [24]. Alternatively, queries can also be expressed in the raw mu-calculus. As the formulas inside the queries are aggregated, hooks from the original queries to the translations are added to the symbol table. In this way a potentially heterogeneous batch of formulas is unified into a single language.

Since the translation from some logics such as CTL^*, which subsumes *LTL*, can require exponential time in the worst case, the front end is augmented with a subsystem that remembers the translation of query forms instantiated multiple times over different choices of atomic propositions. When a closed subexpression is repeated, the system is capable of recalling the translation and simply inserting the current atomic propositions.

Following unification, the queries are then translated and merged into the intermediate language. Called *equation block form*, this is the language over which the optimizer operates. Equation block form is well-understood and can be regarded as a factored representation of the mu-calculus formulas in the same way that a pseudo-assembly language program can be considered a factored representation of an imperative program. Intuitively, each equation in a block system can be regarded as a single logical operation and each block in the system as the encapsulation of a single (possibly trivial) fixed point computation. The translation from the modal mu-calculus to the list of equations that comprise the blocks is a straightforward, syntax-directed process [7]. The equations are then partitioned into their respective blocks as dictated by their interdependencies.

2.2 Intermediate Language

The structure of equation block form is described by the following sequence of definitions.

Definition 1. *Given Var, a finite set of variables each capable of representing a set of model states, AP, a set of atomic propositions that is closed under negation, and Act, a set of actions (or modalities) a* **simple formula***, Φ, is an expression of the form*

$$\Phi ::= p \mid X_1 \mid X_1 \vee X_2 \mid X_1 \wedge X_2 \mid \langle a \rangle X_1 \mid [a] X_1$$

where $p \in AP$, $X_i \in Var$ for $i \in \{1, 2\}$, and $a \in Act$.

For the remainder of this paper, variables defined by the first form will be referred to as *atomic-defined*, variables of the second form as *variable-defined*, variables of the third and fourth forms as *junction-defined*, and variables of the final two forms as *modal-defined*. Variables that appear in a simple formula are said to be *used* by the formula.

Definition 2. *An* **equation block** *has one of two types, $min\langle E \rangle$ or $max\langle E \rangle$, where $\langle E \rangle$ is a non-empty list of equations,*

$$\langle X_1 = \Phi_1, \ldots, X_n = \Phi_n \rangle,$$

each X_i is a distinct element of Var, and each Φ_i is a simple formula.

Over a collection of equation blocks in which no variable is multiply defined, if an equation defines a variable X_i that is used in the simple formula that defines X_j, then X_j *depends* on X_i. If a chain of dependence occurs through zero or more intermediate variables then X_j *transitively depends* on X_i. Variables X_i and X_j are *cyclically dependent* if each transitively depends on the other. These notions extend in the obvious way to describe types of dependence between equation blocks and will be crucial to describing the execution of the various optimizations.

A variable is said to *occur free* in an equation block if the variable is used by a formula of the block but is not defined by any of the equations contained in the block. A set of equation blocks is *closed* if every variable used by an equation in the block set is defined by an equation in the block set.

Definition 3. *An* **equation block system***, $\mathcal{B}^{Var}$, is a closed, ordered list of equation blocks,*

$$\langle B_0, \ldots, B_n \rangle,$$

in which every element of Var is defined by precisely one simple formula.

The *index*, $I(X)$, of a variable, X, in an equation block system is the index of the block in which it is defined. It is assumed that the block order always respects the transitive variable-wise dependence relation except where prohibited by cyclic dependences. More precisely, if X_i depends on X_j, then $I(X_j) \leq I(X_i)$ unless X_j transitively depends on X_i. If X_i and X_j are cyclically dependent then no relationship between $I(X_i)$ and $I(X_j)$ can be enforced without changing the semantics of the equation block system. This assumption is never violated by the standard translation and will be preserved throughout the optimization process. For the remainder of this paper, this is referred to as the *block ordering assumption*. Explicitly stating this assumption makes it easier to reason about the correctness of certain optimizations.

The semantics of an equation block system follow from the semantics for mu-calculus formulas [7]. They can also be inferred from the algorithm in Figure 1 that resolves an equation block system over a Kripke structure [5]. This figure will be referred to again in the next section when discussing how certain optimization passes work. Recall that given a set, AP, of atomic propositions, a *Kripke structure*, K, is a tuple, (S, R, L), where S is a (possibly infinite) set of states, $R \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function that associates with each state the set of atomic propositions that are true in that state. For equation block systems with multiple modalities, each modality, a, is defined by a distinct transition relation, R_a. Hierarchical state machines can be thought of as finite representations of possibly infinite Kripke structures; in this case the semantics are defined to be compatible with the Kripke structure semantics over the full expansion. When the algorithm terminates, each variable is bound to the set of states at which the variable is valid. Termination is guaranteed by the absence of negation among the simple formulas and the block ordering assumption, which ensures monotonicity.

From the semantics, it is evident that the evaluation of a block can result in previous, dependent, blocks being returned to the workset. This behavior occurs within strongly connected components of blocks under the *depends* relation and can continue until the component is fully evaluated.

The *final index*, $F(X)$, of a variable, X, in an equation block system is the maximal index among the set of blocks that are cyclically dependent on the block defining X. Intuitively, once the block indexed $F(X) + 1$ is reached by the semantic algorithm, the set of states bound to X is finalized. If a block is not cyclically dependent with any other block besides itself then the block is referred to as a *singleton*. Singletons are guaranteed to be processed exactly once by the semantic algorithm.

Notice that an equation block system is not generally a unique representation of a mu-calculus formula. In particular, blocks can sometimes be merged, partitioned, and reordered without changing the semantics. Likewise, individual equations can often be simplified or re-expressed. The optimizer will be seen to exploit these facts in Section 3.

Finally, notice that for some blocks, the block type does not affect the semantics of the equation block system. If B_i is a block and for each B_j, B_j depends on B_i implies that $i < j$, then the initial values of the variables defined in B_i are irrelevant to the semantics and the block is said to have *neutral* type. Although not introduced by the standard translation, when this condition is discovered then altering the block type acts as a hint to the optimizer that certain, otherwise unsound, operations are permissible. Semantically, these blocks can be treated as *min* type blocks.

1. Initialize a workset with the set of all blocks and bind each element of Var according to its defining block type, $max \mapsto S, min \mapsto \emptyset$.
2. Remove the least indexed block, B_i, from the workset and process it as follows:
   (a) Save the current bindings for each variable defined in the block.
   (b) Update the bindings by iteratively solving the block equations until a fixed point is reached using the current bindings for the variables that occur free in the block. Each equation $X = \Phi$ is evaluated as follows:

$$s \in X \text{ iff } \begin{cases} \Phi = p \in AP \text{ and } p \in L(s) \\ \Phi = X_1 \text{ and } s \in X_1 \\ \Phi = X_1 \vee X_2 \text{ and } s \in X_1 \cup X_2 \\ \Phi = X_1 \wedge X_2 \text{ and } s \in X_1 \cap X_2 \\ \Phi = \langle a \rangle X_1 \text{ and } \exists t \in X_1 . R_a(s,t) \\ \Phi = [a] X_1 \text{ and } \forall t \in S . R_a(s,t) \rightarrow t \in X_1 \end{cases}$$

   (c) For each $k \neq i$ such that there is a variable defined in B_k that transitively depends on a variable defined in B_i whose binding has changed from its saved binding (a), add B_k to the workset and if $k < i$ reset each defined variable in B_k according to its block type, $max \mapsto S, min \mapsto \emptyset$.
3. Repeat the previous step (2) until the workset is empty.
4. Return the variable bindings as the solution.

Figure 1: Algorithm for Resolving an Equation Block System, $\mathcal{B}^{Var}$, against a Kripke Structure, $K = (S, R, L)$

2.3 Symbol Table

Intuitively, each variable defined in an equation block system corresponds to some sub-expression of an original mu-calculus formula. Likewise, each equation block system contains one variable that corresponds to each entire formula. It is this *top* variable whose validity is of primary concern — the set of states ultimately bound to this variable is the set of states for which that mu-calculus formula is valid. It is in this sense that the mu-calculus and equation block form are considered equivalent logics.

During the translation process, the equation block systems arising from a batch of formulas are concatenated into a single list of equation blocks, with the variables from each suitably renamed to ensure uniqueness and avoid conflicts. Since each original formula is closed, this does not violate the block ordering assumption. During the compilation process the equation block system will be rearranged. A symbol table is used to track information about each variable. The information recorded for each variable consists of:

• Index	• Final Index
• Defining Block Type	• Defining Block a Singleton?
• Defining Formula	• Top Variable?
• List of Uses	• Index of Next Block

This information is initialized during the translation of each formula and is updated throughout the compilation process. Additionally, each top variable is bound to a supplemental record containing information from the original query, including its source logic. This information can also include a textual description of the query, a list of parameters, and possibly, instructions for post-processing. The list of *top* variables is used throughout the compilation process to decide what information it is necessary to retain in the equation block system and what information it is safe to discard. After compilation, information stored in the table is used to generate additional execution instructions for the model checking engine.

In the example of Figure 2, the zero-subscripted variables are the *top* variables and each would be bound to a record containing the original formula and logic type (CTL). As an example, the initial symbol table entries for the variable x_4 would look like:

Field	Value	Field	Value
Index	0	Final Index	0
Type	min	Singleton?	$true$
Formula	$x_5 \wedge x_6$	Top?	$false$
Uses	$\{x_2\}$	Next Index	1

2.4 Back End

The Carnauba model checking engine is capable of checking raw equation blocks over unrestricted hierarchical state machines, so only minimal additional back end translation is required. Specifically, any cyclic dependencies among the variable and junction-defined variables are eliminated. While such cyclic dependences can occur as a consequence of the standard translation, an equivalent system can always be produced without these cyclic dependences [4]. Further, the symbol table is used to generate a script instructing the engine which variables to report and how to use the result to determine for which variables to generate an example.

More sophisticated back ends are possible, including those that produce specialized forms, such as Büchi automata. While equation block form is more expressive than some common temporal logics, many block patterns can be translated back into other logics. The ability to substitute optimization passes that preserve these forms makes the framework compatible with other model checking systems based on specific logics, such at LTL. In this way, the framework is compatible with other optimization techniques.

3. THE OPTIMIZER

The logic compiler optimizer is organized as a system of independent optimization routines. As in a traditional compiler, each routine embodies a single optimization concept. In fact, many of the optimizations applicable to equation block systems are inspired by traditional programming language optimizations. In this section, each of the eight basic optimizations are outlined in some detail. This is followed by a discussion of the pass scheduler. Finally, a comprehensive example is presented to illustrate how the passes come together to produce an effective result.

One novelty of this compiler is that it is capable of operating in either a model-independent or model-dependent mode. The choice of mode affects the operation of the confluence and comparison operations over the modalities and atomic propositions. In the model-independent mode, modalities and atomic propositions are treated as abstract. In this mode, the optimizer is restricted to the syntactic notion of equality. Comparisons between distinct abstract modalities or atomic propositions are always otherwise inconclusive. Union and intersection operations generate new abstractions that embody information about how they were constructed. In this way, equality and containment can still be inferred if two abstract objects have appropriate constructions. This abstraction is augmented by the inclusion of *absolute constants.* The atomic propositions *true*and *false*are used to denote the full and empty set of model states, respectively. The modality *true*, also denoted in the syntax by the absence of an explicit modality, likewise refers to the full set of model transitions. In the model-dependent mode, comparisons between modalities or atomic propositions involve a complete inspection of the set of transitions or states that are represented. The model-dependent mode also records whether every model state has at least one successor under the *true*modality and uses this during the optimization process. This mode, therefore, subsumes the optimizations performed in the model-independent mode.

3.1 Optimization Routines

There are eight distinct optimization routines:

1. **Dead Equation Elimination**
2. **Perform Generic Solve**
3. **Unify Atomic Propositions**
4. **Normalize Equation Block System**
5. **Remove Trivial Equations**
6. **Simplify Redundant Junctions**
7. **Sever Isomorphic Expressions**
8. **Perform Peephole Substitutions**

The overarching goal of the optimization routines is to reduce the complexity of the equation block system without altering its semantics. Complexity can be approximated by the number of variables occurring in the system and their formula type. Modal-defined variables tend to increase the running time of a model checker; they drive the non-trivial fixed point computations. Atomic-defined variables tend to increase the space required. The optimizations are constructed so that the semantics of each variable is preserved, including non-*top* variables, unless the variable is completely eliminated from the system. In select cases, new variables are added to the system to take the place of more complex subsystems of variables. The optimizations are guaranteed to retain each of the *top* variables in the system; the variables corresponding to the original query formulas. Optimizations that preserve the semantics of each of the *top* variables are said to be *correct.* Finally, notice that the number of edges in the *depends* relation is bounded above by twice the number of variables defined in the system. Therefore, all of the complexity measures can be described in terms of numbers of variables.

3.1.1 Dead Equation Elimination

This is a straightforward optimization based on dead code elimination [18] in a traditional compiler. In a pass of this routine, equations that do not contribute to the solution of one or more of the *top* variables in the equation block system are eliminated. This pass is used to sweep away parts of the equation block system that have been found to be redundant or unnecessary by other passes in the optimizer.

The optimization starts by marking the equations defining the *top* variables and proceeds by depth-first search along the reverse of the *depends* relation. Equations that define variables needed in the definition of the *top* variables are ultimately marked. When the search terminates, any unmarked equations are eliminated in place from the system. A final sweep then eliminates any empty equation blocks and updates the indexes stored in the symbol table. The preservation of the semantics is guaranteed by the nature of the *depends* relation; no variable that is transitively required by the algorithm of Figure 1 in the evaluation of a *top* variable is eliminated. This optimization does not disturb the ordering of the remaining variables and requires time linear in the number of variables.

3.1.2 Perform Generic Solve

This optimization is partially inspired by the notion of constant propagation [26] for imperative programs. The goal is to propagate inward the absolute atomic constants while using them to simplify modal and junction formulas. Additionally, variables defined entirely by atomic propositions with no intervening modal expressions are redefined as atomic-defined variables via unions and intersections of those atomic propositions. These are constructed explicitly in the model-dependent case, while the operators work abstractly in the model-independent case.

Propagation is accomplished by running the algorithm described in Figure 1, but with the evaluation rules in step 2b replaced with those of Figure 4. Under the new evaluation rules, a value of zero (0) can be interpreted as *"will never know"*. Similar to the third value in three-value logic, it denotes a variable whose solution cannot be simplified to an atomic proposition without evaluating a non-trivial modal expression. Essentially, these rules generate the most complete solution available without knowing the precise transition relation of the model. The rules in the figure utilize the assumption that every state has a successor under the *true* modality. If this assumption is invalid or cannot be determined, then the pass proceeds by using the rules for other modalities even in the $a = \mathit{true}$ case.

When the generic solve terminates the non-zero bindings are used to transform the equation block system. Variables are considered in order of increasing index. For a variable, X, defined in a singleton block found to have non-zero binding p, if $p = \mathit{true}$ then equations of the form $Z = X \wedge Y$ become $Z = Y$. If $p = \mathit{false}$ then equations of the form $Z = X \vee Y$ become $Z = Y$. Finally, the equation defining X becomes $X = p$. For variables defined in non-singleton blocks, the junction transformations are restricted to cases where $F(X) < I(Z)$. Also, instead of redefining X at its current index, a new *neutral* type block with index $F(X) + 1$ is created containing the single equation $X = p$. Every instance of X in B_k, $k \leq F(X)$ in then converted to a new variable name. The symbol table entries for indexes, formulas, and uses are updated as each variable is considered.

The correctness of this optimization follows from the evaluation rules used for this routine that guarantee that the final binding of each variable is either its actual solution or

[**Query 1**] **CTL** : **EX EF p** $\mapsto x_0$ in

$$\text{min}\ \{\quad x_0 = \Diamond x_1 \quad x_1 = x_2 \quad x_2 = x_3 \vee x_4 \quad x_3 = p \quad x_4 = x_5 \wedge x_6 \quad x_5 = \mathit{true} \quad x_6 = \Diamond x_1 \quad \}$$

[**Query 2**] **CTL** : **EX EF EF p** $\mapsto z_0$ in

$$\begin{array}{llllllll}
\text{min}\ \{ & z_0 = \Diamond z_1 & z_1 = z_2 & z_2 = z_3 \vee z_9 & z_3 = z_4 & z_4 = z_5 \vee z_6 & z_5 = p & z_6 = z_7 \wedge z_8 \\
 & z_7 = \mathit{true} & z_8 = \Diamond z_3 & z_9 = z_{10} \wedge z_{11} & z_{10} = \mathit{true} & z_{11} = \Diamond z_1 & \} &
\end{array}$$

[Pass 1] Perform Generic Solve: $x_4 \mapsto x_6$, $z_6 \mapsto z_8$, and $z_9 \mapsto z_{11}$

$$\begin{array}{lllllllll}
\text{min}\ \{ & x_0 = \Diamond x_1 & x_1 = x_2 & x_2 = x_3 \vee x_4 & x_3 = p & x_4 = x_6 & x_5 = \mathit{true} & x_6 = \Diamond x_1 & \} \\
\text{min}\ \{ & z_0 = \Diamond z_1 & z_1 = z_2 & z_2 = z_3 \vee z_9 & z_3 = z_4 & z_4 = z_5 \vee z_6 & z_5 = p & z_6 = z_8 & z_7 = \mathit{true} \\
 & z_8 = \Diamond z_3 & z_9 = z_{11} & z_{10} = \mathit{true} & z_{11} = \Diamond z_1 & \} & & &
\end{array}$$

[Pass 2] Dead Equation Elimination: $\{x_5, z_7, z_{10}\}$ eliminated

$$\begin{array}{llllllll}
\text{min}\ \{ & x_0 = \Diamond x_1 & x_1 = x_2 & x_2 = x_3 \vee x_4 & x_3 = p & x_4 = x_6 & x_6 = \Diamond x_1 & \} \\
\text{min}\ \{ & z_0 = \Diamond z_1 & z_1 = z_2 & z_2 = z_3 \vee z_9 & z_3 = z_4 & z_4 = z_5 \vee z_6 & z_5 = p & z_6 = z_8 \\
 & z_8 = \Diamond z_3 & z_9 = z_{11} & z_{11} = \Diamond z_1 & \} & & &
\end{array}$$

[Pass 3] Unify Atomic Propositions: $\{x_3, z_5\} \mapsto x_3$

$$\begin{array}{llllllll}
\text{min}\ \{ & x_0 = \Diamond x_1 & x_1 = x_2 & x_2 = x_3 \vee x_4 & x_3 = p & x_4 = x_6 & x_6 = \Diamond x_1 & \} \\
\text{min}\ \{ & z_0 = \Diamond z_1 & z_1 = z_2 & z_2 = z_3 \vee z_9 & z_3 = z_4 & z_4 = z_5 \vee z_6 & z_5 = x_3 & z_6 = z_8 \\
 & z_8 = \Diamond z_3 & z_9 = z_{11} & z_{11} = \Diamond z_1 & \} & & &
\end{array}$$

[Pass 4] Normalize Equation Block System

$$\begin{array}{llllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_1 = x_2 & x_2 = x_3 \vee x_4 & x_4 = x_6 & x_6 = \Diamond x_1 \quad \} \\
\text{neutral}\ \{ & z_5 = x_3 & \} & \text{min}\ \{ & z_3 = z_4 & z_4 = z_5 \vee z_6 & z_6 = z_8 & z_8 = \Diamond z_3 \quad \} \\
\text{min}\ \{ & z_1 = z_2 & z_2 = z_3 \vee z_9 & z_9 = z_{11} & z_{11} = \Diamond z_1 & \} & & \\
\text{neutral}\ \{ & x_0 = \Diamond x_1 & \} & \text{neutral}\ \{ & z_0 = \Diamond z_1 & \} & &
\end{array}$$

[Pass 5] Remove Trivial Equations: $\{x_1, x_4, z_1, z_3, z_5, z_6, z_9\}$ trivially defined

$$\begin{array}{lllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_2 = x_3 \vee x_6 & x_6 = \Diamond x_2 & \} \\
\text{min}\ \{ & z_4 = x_3 \vee z_8 & z_8 = \Diamond z_4 & \} & \text{min}\ \{ & z_2 = z_4 \vee z_{11} & z_{11} = \Diamond z_2 \quad \} \\
\text{neutral}\ \{ & x_0 = \Diamond x_2 & \} & \text{neutral}\ \{ & z_0 = \Diamond z_2 & \} &
\end{array}$$

[Pass 6] Simplify Redundant Junctions: $z_{11} \to z_4$ so $z_2 = z_4 \vee z_{11} \mapsto z_2 = z_4$

$$\begin{array}{lllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_2 = x_3 \vee x_6 & x_6 = \Diamond x_2 & \} \\
\text{min}\ \{ & z_4 = x_3 \vee z_8 & z_8 = \Diamond z_4 & \} & \text{min}\ \{ & z_2 = z_4 & z_{11} = \Diamond z_2 \quad \} \\
\text{neutral}\ \{ & x_0 = \Diamond x_2 & \} & \text{neutral}\ \{ & z_0 = \Diamond z_2 & \} &
\end{array}$$

[Pass 7] Dead Equation Elimination: $\{z_{11}\}$ eliminated,
[Pass 8] Normalize Equation Block System,
[Pass 9] Remove Trivial Equations: $\{z_2\}$ trivially defined

$$\begin{array}{lllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_2 = x_3 \vee x_6 & x_6 = \Diamond x_2 & \} \\
\text{min}\ \{ & z_4 = x_3 \vee z_8 & z_8 = \Diamond z_4 & \} & & & \\
\text{neutral}\ \{ & x_0 = \Diamond x_2 & \} & \text{neutral}\ \{ & z_0 = \Diamond z_4 & \} &
\end{array}$$

[Pass 10] Sever Isomorphic Expressions: $\{x_0, z_0, x_6, z_8\} \mapsto x_6$

$$\begin{array}{lllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_2 = x_3 \vee x_6 & x_6 = \Diamond x_2 & \} \\
\text{min}\ \{ & z_4 = x_3 \vee z_8 & z_8 = x_6 & \} & & & \\
\text{neutral}\ \{ & x_0 = x_6 & \} & \text{neutral}\ \{ & z_0 = x_6 & \} &
\end{array}$$

[Pass 11] Dead Equation Elimination: $\{z_4, z_8\}$ eliminated,
[Pass 12] Normalize Equation Block System,
[Pass 13] Remove Trivial Equations: $\{x_0, z_0\}$ trivially defined

$$\begin{array}{lllllll}
\text{neutral}\ \{ & x_3 = p & \} & \text{min}\ \{ & x_2 = x_3 \vee x_0 & x_0 = \Diamond x_2 & \} \\
\text{neutral}\ \{ & z_0 = x_0 & \} & & & &
\end{array}$$

Figure 2: Sequence of Optimization Passes on a Two-Query Batch Example

Pass	Possibly Reschedules							
	DEE	PGS	UAP	NEBS	RTE	SRJ	SIE	PPS
DEE								
PGS	●		●	●	●	●	●	●
UAP								
NEBS								
RTE								
SRJ	●	●	●	●	●		●	●
SIE	●			●	●			
PPS	●			●	●			

Figure 3: Interdependencies between Optimization Passes

Binary Operators

$\vee$	*true*	*false*	q	0
true	*true*	*true*	*true*	*true*
false	*true*	*false*	q	0
p	*true*	p	$p \cup q$	0
0	*true*	0	0	0

$\wedge$	*true*	*false*	q	0
true	*true*	*false*	q	0
false	*false*	*false*	*false*	*false*
p	p	*false*	$p \cap q$	0
0	0	*false*	0	0

Atomic Propositions, Variables, & Unary Operators

X	q	X	$\Diamond X$	$\langle a \rangle X$	$\Box X$	$[a]X$
true	q	*true*	*true*	0	*true*	*true*
false	q	*false*	*false*	*false*	*false*	0
p	q	p	0	0	0	0
0	q	0	0	0	0	0

Assume that $a \neq \mathit{true}$.

Figure 4: Evaluation Tables for Equation Forms in Generic Solve

zero. Only those variables solved to non-zero values have their expressions replaced in the system and only at the point at which their value may be considered finalized. Since no cyclic dependence is destroyed and no reordering takes place, the block ordering assumption is also preserved. For any fixed alternation depth, d, each variable needs to be updated at most 3^d times. Hence, the running time is linear in the number of variables for any bounded alternation depth.

3.1.3 Unify Atomic Propositions

In this optimization, an equivalence relation is constructed among the atomic-defined variables in the equation block system using the notion of atomic proposition equivalence directed by the optimization mode. For each non-trivial equivalence class in this relation, the variable with the least index becomes the class representative. In the event of a draw, a representative is chosen arbitrarily from among the set of least indexed variables. Each other variable in the equivalence class is then redefined in its current block as the class representative. The symbol table entries are updated for each variable in the equivalence class. When this pass terminates, each atomic proposition in the system is unique with respect to the predetermined notion of equivalence.

The correctness of this optimization follows from the choice of the least-indexed member of each equivalence class as the class representative. Atomic-defined variables cannot be part of a cyclic dependence. By the block ordering assumption they can be considered finalized before they are ever used. The least indexed member of an equivalence class can then be considered finalized before any of the new variable equations are evaluated. Thus, the semantics of the system are preserved.

The purpose of this optimization is three-fold. First, it minimizes the number of distinct atomic propositions that occur in the equation block system. For a model checker, this reduces the number of distinct sets of model states that must be manipulated. Second, although it does not reduce the number of variables in the system, it does introduce trivial equations that subsequent passes can eliminate, thus reducing the size of the equation block system. Finally, it eliminates the need for any further comparisons among the atomic-defined variables. In the model-dependent case, these comparisons can be non-trivial, and performing them all in a single pass allows subsequent passes to require only the trivial notion of syntactic equality as the test in either optimization mode. The performance of this optimization is bounded by the time to create the equivalence relation. In the worst case, where there are no equivalences, the number of required comparisons is quadratic in the number of atomic-defined variables.

3.1.4 Normalize Equation Block System

This routine does not perform any specific optimization. Rather, it transforms the equation block system into a form that both facilitates other optimizations as well as makes model checking more efficient. First, it repartitions the individual equations to make the equation blocks as small as possible, while at the same time minimizing the number of times the semantic algorithm of Figure 1 would need to reprocess any particular block. Second, it identifies blocks whose type can be changed to *neutral*. This designation speeds up subsequent optimization and provides a possible hint to the model checking engine that the block specified fixed point can be computed directly without iteration.

This optimization again relies heavily on the *depends* relation. First, the existing equation blocks are dissolved in place to a list of equations, each tagged with their original block type and index. This list of equations is then partitioned into a topologically sorted list of strongly connected components via the *depends* relation. Within each component, the equations are partitioned again by their nesting depth; the maximal number of type alternations that occur on a simple dependence cycle consisting only of equations at or below the tagged index of the equation. The classes are ordered from least to greatest and each is then partitioned a final time into a topologically sorted list of strongly connected components using the same *depends* relation restricted to the equivalence class. These components, which must consist of equations all of the same type, are the blocks. Neutral blocks are identified during this final partitioning. The blocks are then concatenated together, in order, and the symbol table is reconstructed. Computing the strongly connected components requires $O(n \log n)$ time in the number of variables. Computing the nesting depths requires quadratic time over the variables in each component.

3.1.5 Remove Trivial Equations

A trivial equation is a variable-defined equation that is unnecessary to the semantics of the equation block system. Trivial equations occur as a natural by-product of the translation to equation block form as well as from other optimizations. For example, recall that *Unify Atomic Propositions* replaces redundant atomic propositions with trivial equations. This optimization seeks to remove these equations by unifying both sides of the equation into a single variable.

Processing occurs by examining each equation in each

Source Pattern: s_0 is semantically equivalent to $CTL : A[P\ U\ Q]$.

$$\begin{array}{lllllll}
\text{neutral } \{ & s_3 = P & & \} & \text{neutral } \{ & s_4 = Q & \} \\
\text{max } \{ & s_1 = s_3 \wedge s_5 & s_5 = s_4 \vee s_6 & s_6 = \Box s_1 & \} & & \\
\text{min } \{ & s_2 = s_4 \vee s_7 & s_2 = \Box s_2 & \} & \text{neutral } \{ & s_0 = s_1 \wedge s_2 & \}
\end{array}$$

Target Pattern: t_0 is semantically equivalent to s_0.

$$\begin{array}{llllll}
\text{neutral } \{ & t_1 = Q & & \} & \text{neutral } \{ & t_3 = P \quad \} \\
\text{min } \{ & t_0 = t_1 \vee t_2 & t_2 = t_3 \wedge t_4 & t_4 = \Box t_0 & \} &
\end{array}$$

Original Code: z_0 is semantically equivalent to $CTL : A[(AG\ W)U\ (AG\ X)]$.

$$\begin{array}{llllllll}
\text{neutral } \{ & z_8 = W & \} & \text{neutral } \{ & z_{10} = X & \} & & \\
\text{max } \{ & z_3 = z_8 \wedge z_9 & z_9 = \Box z_3 & \} & \text{max } \{ & z_4 = z_{10} \wedge z_{11} & z_{11} = \Box z_4 & \} \\
\text{max } \{ & z_1 = z_3 \wedge z_5 & z_5 = z_4 \vee z_6 & z_6 = \Box z_1 & \} & & & \\
\text{min } \{ & z_2 = z_4 \vee z_7 & z_7 = \Box z_2 & \} & \text{neutral } \{ & z_0 = z_1 \wedge z_2 & \} &
\end{array}$$

Optimized Code: Pattern matches $s_0 \cong z_0 \cong t_0$, $s_3 \cong z_3 \cong t_3$, and $s_4 \cong z_4 \cong t_1$.

$$\begin{array}{llllllll}
\text{neutral } \{ & z_8 = W & \} & \text{neutral } \{ & z_{10} = X & \} & & \\
\text{max } \{ & z_3 = z_8 \wedge z_9 & z_9 = \Box z_3 & \} & \text{max } \{ & z_4 = z_{10} \wedge z_{11} & z_{11} = \Box z_4 & \} \\
\text{min } \{ & t_0 = t_1 \vee t_2 & t_1 = z_4 & t_2 = t_3 \wedge t_4 & t_3 = z_3 & t_4 = \Box t_0 & \} & \\
\text{max } \{ & z_1 = z_3 \wedge z_5 & z_5 = z_4 \vee z_6 & z_6 = \Box z_1 & \} & & & \\
\text{min } \{ & z_2 = z_4 \vee z_7 & z_7 = \Box z_2 & \} & \text{neutral } \{ & z_0 = t_0 & \} &
\end{array}$$

Dead Code Eliminated: $\{z_1, z_2, z_5, z_6, z_7\}$ eliminated

$$\begin{array}{llllllll}
\text{neutral } \{ & z_8 = W & \} & \text{neutral } \{ & z_{10} = X & \} & & \\
\text{max } \{ & z_3 = z_8 \wedge z_9 & z_9 = \Box z_3 & \} & \text{max } \{ & z_4 = z_{10} \wedge z_{11} & z_{11} = \Box z_4 & \} \\
\text{min } \{ & t_0 = t_1 \vee t_2 & t_1 = z_4 & t_2 = t_3 \wedge t_4 & t_3 = z_3 & t_4 = \Box t_0 & \} & \\
\text{neutral } \{ & z_0 = t_0 & \} & & & & &
\end{array}$$

Figure 5: Example of Perform Peephole Substitutions

block, in order. Specifically, a trivial equation is defined to be an equation of the form $X = Y$ in which X does not occur free in any B_k with $k < I(X)$. In these cases, the equation is removed and every use of X is converted to a use of Y. In the event that an equation of the form $X = X$ is created, then this is replaced by the equation $X = p$ where $p = \mathit{true}$ if the block has type *max* and $p = \mathit{false}$ otherwise. If X is a top variable and Y is not, then X is substituted for Y throughout the entire equation block system. If both X and Y are top variables, then a new *neutral* block consisting of the single equation, $X = Y$, is appended to the end of the equation block system. This ensures that each top variable is preserved in the system.

Correctness is ensured by the restrictions placed on the variable equations that can be eliminated. Only those variables whose initial value is never required have their trivial equation eliminated. Hence, the semantics of the system is preserved. It is straightforward to check that this optimization cannot violate the block ordering assumption. This optimization requires linear time.

3.1.6 *Simplify Redundant Junctions*

This optimization examines each junction equation to determine if there is an implication relationship between the two operands that holds each time the equation is evaluated. If such an implication relationship exists, it can be used to simplify the formula to either a single operand or an absolute atomic constant. The tests and simplifications for each junction form are as follows:

Equation	*Test Always?*	*Reduction*
$Z = X \vee Y$	$X \rightarrow Y$	$Z = Y$
	$Y \rightarrow X$	$Z = X$
	$\neg X \rightarrow Y$	$Z = \mathit{true}$
$Z = X \wedge Y$	$X \rightarrow Y$	$Z = X$
	$Y \rightarrow X$	$Z = Y$
	$X \rightarrow \neg Y$	$Z = \mathit{false}$

Each test involves attempting to prove by induction that the implication rule between the two variables holds universally. This requires testing both the base case (first invocation) and using this as an inductive hypothesis for testing each subsequent invocation initiated by the fixed point semantics. To accomplish this, both variables used in the junction formula are first expanded into a DNF expression in which atomic propositions, modal expressions, and higher-indexed variables make up the terminals. The proof then proceeds by trying to satisfy a sufficient (though not necessary) condition for the implication of one DNF form by another. Specifically, DNF expression $\mathcal{E}_1$ implies $\mathcal{E}_2$ if for every conjunctive clause, $\mathcal{C}_i$ in $\mathcal{E}_1$ there exists $\mathcal{C}_j$ in $\mathcal{E}_2$ such that $\mathcal{C}_i \rightarrow \mathcal{C}_j$. Likewise, $\mathcal{C}_i \rightarrow \mathcal{C}_j$ if for every atom a_l in $\mathcal{C}_j$ there exists a_k in $\mathcal{C}_i$ such that $a_k \rightarrow a_l$. The proof proceeds by recursive descent. Atomic proposition p implies q if $p \subseteq q$. Modal expression $[a]X$ implies $[b]Y$ if the expressions are from the same block type, $a \subseteq b$, and X recursively implies Y. The base test substitutes the initial values for the terminals while the recursive test carries the previous test as an assumption. The process terminates when an implication is found or when every possible matching has been exhausted. Any test on the formula defining X that would result in the elimination of a use of variable Y fails trivially if $I(Y) < I(X) \leq F(Y)$. A quick inspection of the DNF forms for mismatched modalities or atomic propositions before testing precludes most failing tests.

This test is a sufficient, though not necessary, condition to prove that the appropriate implication is a tautology. Since this optimization can remove uses of some variables, it can potentially result in dead variables that can removed by subsequent passes of *Dead Equation Elimination*.

The trivial failure test precludes eliminating any dependence that would violate the block ordering assumption. The correctness of this optimization follows form the fact that the implication holds each time the junction formula is evaluated. As a result, the simplifications do not change the value derived by the formula for any evaluation and hence do not change the semantics of the equation block system. The performance of this optimization is bounded by the time to perform the tests. In the worst case, the test used in this implementation is quadratic in the number of variables on which the junction operands transitively depend.

3.1.7 *Sever Isomorphic Expressions*

This optimization builds equivalence classes by identifying pairs of isomorphic variables in the system. Isomorphic variables are variables that are provably equivalent at the termination of the semantic algorithm. Given these equivalence classes, the variable X with the least $F(X)$ in the class becomes the representative. Each other singleton, Y, in the class is redefined, in place, by the variable equation, $Y = X$. For other members of the class, Z, a new *neutral* block is created at index $F(Z)+1$ with the equation $Z = X$ and every instance of Z in a block B_k with $k \leq F(Z)$ is renamed to a new name. The result is that equations defining an isomorphic copy of an expression become "severed" as there are no longer any uses. *Dead Equation Elimination* is then able to discard these equations. The symbol table is updated as each equation is modified or created.

Equivalence is determined using an implication proof technique similar to the one used in *Simplify Redundant Junctions*. In this case $X \cong Y \leftrightarrow (X \rightarrow Y) \wedge (Y \rightarrow X)$. The complexity of this optimization comes from the need to determine which variable pairs to test. To achieve a maximal result, it is unnecessary to find every isomorphism that exists; one isomorphism can result in the elimination of several variables with isomorphic partners. Testing for these isomorphic partners would be redundant. Further, it is computationally infeasible to test for an isomorphism between every pair of variables. To minimize the number of tests, a signature is computed for each variable consisting of the atomic propositions and modal operators on which the variable transitively depends. This signature is intended to be a compelling precondition for equivalence. The signature can be computed quickly and only variables with the same signature need to be compared. As each new variable is added to the equivalence relation within a particular signature, it is only tested against one representative for each class so far identified within that signature. Further, to prevent redundant testing, tests are restricted to the *top* variables and variables used at an index strictly greater than their own index. From these equivalence classes, equivalences between the remaining variables are restricted to structural isomorphism. For instance, if $X = [a]W$, $Z = [a]Y$, and it was found that $W \cong Y$, then $X \cong Z$.

The correctness of the optimization follows from the restriction that a variable can only be redefined to an equivalent variable that is immutable before the first evaluation of the replacement formula. In essence, this replaces the variable with its solution which does not change the semantics of the equation block system. The running time is dependent on the number of tests that must be performed. In the worst case, this is quadratic in the number of variables within each signature.

3.1.8 *Perform Peephole Substitutions*

This is an optional optimization inspired by the notion of peephole optimizers for low level machine code [20]. The preceding passes break the problem of reducing the complexity of an equation block system into the execution of small, semantics-preserving transformations. However, cases can arise in practice when an equation pattern is present for which there exists a simpler equivalent pattern, but for which there does not exists a simple sequence of transformations taking the more complex pattern to the simpler one. This routine fills the need to perform this transformation by providing a library of "large scale" translation rules.

The peephole analysis works by comparing variables to entries in a library of pattern transformations in an attempt to find a syntactic equivalence. When such an equivalence is found the variable is redefined as the top variable in a corresponding target pattern. The closed expression variables of the target pattern are then instantiated based on the syntactic equivalence with the source pattern. The necessary variables for the target pattern are added to the equation block system so as not to conflict with any existing variable names. The intent is that a subsequent pass of *Dead Equation Elimination* will sweep the variables of the original source pattern from the system. Figure 5 shows an example of a peephole optimization that could not be performed by any combination of the preceding passes.

Finally, while it is true that an entire optimization system could be based around the application of a library of reduction rules, such a system would be inefficient. The intent of this pass is not to perform simple optimizations but rather to provide a tool within the framework for performing certain domain-specific, monolithic transformations.

3.2 Dynamic Scheduler

As in any optimizing compiler, the sequencing of the passes is crucial to efficiency of the system. Rather than using a fixed optimization sequence, the logic compiler uses a dynamic pass scheduler within the static optimization order introduced at the outset of this section. Each optimization routine has an associated flag to indicate if it is scheduled to run. At each iteration, the first optimization in the fixed order that is flagged is executed. When a pass performs a transformation that may open an opportunity for another optimization to make progress, it simply sets the flag for the corresponding routine. A fixed order for the routines was chosen so that later routines could be accelerated by making assumptions that earlier ones had run. For example, the comparison of signatures in *Sever Isomorphic Expressions* is accelerated by the unification of the atomic propositions in *Unify Atomic Propositions*. If earlier optimizations are omitted then the later optimizations are still correct; they are just somewhat less effective.

When the optimizer starts, the flag for each routine is set with the exception of the flag for *Dead Equation Elimination*. Since this is the first optimization in the order, and since the native translation never produces equations that are initially dead, an initial pass of this optimization is unnecessary. *Dead Equation Elimination*, *Unify Atomic Propositions*, *Normalize Equation Block System*, and *Remove Trivial Equations* are essentially cleanup routines and never schedule any additional optimization. The remaining four optimizations set the flag of one or more of the remaining routines whenever they make progress. Figure 3 shows the scheduling interdependencies of the eight optimizations.

3.3 A Comprehensive Example

A complete example of the execution of the optimizer is provided in Figure 2. In the example, two related but independent *CTL* formulas are translated into a single equation block system and optimized. The first query, translated into the x-subscripted variables, should be read: "There exists a successor such that there exists a path where in the future p holds". The second query, translated into the z-subscripted variables, should likewise be read: "There exists a successor such that there exists a path where in the future there exists a path where in the future p holds". Intuitively, the two queries are equivalent; if in the future it is the case that something will hold in the future then it is the case that something will hold in the future. The second query uses a redundant *EF* operator. These queries were chosen to provide a small example that would provide grist to each of the active optimization passes. For this example *Perform Peephole Substitutions* has been turned off; an example of its execution is provided separately in Figure 5.

The queries in the example are translated into equation block form using the standard translations for *CTL* and the modal mu-calculus. Here, both queries use the same syntactic atomic proposition, so the compiler generates the same output in both the model-dependent and model-independent modes. The variables x_0 and z_0 are the *top* variables in this case. The final *neutral* block in the output is required to ensure that both of these variables remain in the system. Notice that the optimizer not only finds the equivalence between the two queries, but that it significantly optimizes the first query as well. In fact, for the first query there is no equivalent *CTL* expression that translates into a block form under the standard translation with only one modal expression. The factored intermediate representation admits an optimization that is impossible at the *CTL* source level. Some other systems compensate for this by introducing extended operators into the native logic that are then interpreted as this optimized form. It is self-evident that the optimized equation block system would place appreciably less demand on a model checking engine than the unoptimized version and that this reduction would be orthogonal to any reduction achieved by reducing the state space of the model over which the resulting equation block system were checked. Since both the optimized and unoptimized versions of the queries can be resolved in time linear in the size of the model, the reduction in model checking time as a percentage is invariant in the size of the model. Experimentally, the optimized version requires less than 30% of the running time of the unoptimized version over a fixed model.

4. EXPERIMENTAL RESULTS

Testing the efficacy of the logic compiler was done by applying it to batches of queries automatically generated from several benchmark programs. Generation of a query batch for each program was done by instantiating a collection of twelve query templates, one at a time, over a set of appropriate program elements. Query templates are Carnauba's way of abstracting common query patterns away from the specific atomic propositions to which they are applied and the temporal logic used to implement them. In the experiment, templates were chosen to produce a uniform, domain non-specific, set of queries of the same sort that would be relevant to program verification for any C program. The twelve templates used, and the program elements over which they were instantiated, are described in Figure 6.

The atomic propositions for each query were generated using Carnauba's system for abstract interpretation to determine when a variable is defined, used, dereferenced, tested, etc. This abstract interpretation is based upon a pointer analysis [21] and accompanying alias analysis that can track effects through intervening variables. Restricting the instantiation to the global variables allowed for the generation of reasonably-sized batches of queries with a diverse collection of atomic propositions. It should be noted that not all queries were applicable to all of the variables or programs to which they were applied. No effort was made to prevent the inclusion of useless queries; it was left to the optimizer to detect and eliminate them.

The compiler is implemented entirely in STk Scheme. It is part of the Carnauba model checking system, developed as a plug-in to GrammaTech's CodeSurfer® [8] program analysis product. All of the codes are C programs taken form the SPEC95 and SPEC2000 benchmark suites. The performance results are presented in Figure 7. All of the results were generated by running the compiler in its model-dependent mode with the optional peephole optimization routine disabled.

The results are organized as follows: The first two columns give the name of the program and the number of queries that were generated for it. *Vars* refers to the number of variables in the equation block system. *Atomic* and *Modal* give the number of atomic- and modal-defined variables in the system. The CPU running time of the optimizer is presented in seconds under *Time*. The total number of optimization passes run and the percentage running-time reduction achieved for the Carnauba model checking system are given in the final two columns. The performance numbers were generated using a 2.0 GHz Pentium 4 PC with 1.0 GB of RAM running CodeSurfer 1.9p2 for Windows XP.

The results show that the optimizer was effective at reducing the size of the equation block system. In the experiments, queries originating from three distinct source logics were merged into the common intermediate representation. In each of the dozen examples, the optimizer reduced both the total number of modal-defined variables (which largely govern the running time) and the number of distinct atomic propositions (which has implications on the space requirement). In all cases, the reduced model checking time significantly exceeded the compilation overhead.

In addition to picking up common sub-expressions within the query batch, the optimizer also eliminated the irrelevant sub-expressions. For example, if a particular pointer variable in a benchmark was never used to store the result of a malloc, as is common, then the malloc and free atomic propositions for the variable would be equivalent to *false*. In this case, *Perform Generic Solve* would use this to essentially reduce the expressions based on these atomic propositions to an atomic constant. In this way the optimizer automatically "cleans-up" irrelevant query sub-expressions without burdening the model checker. This accounts for the larger reduction for the more pointer-based codes. The ability to propagate atomic propositions that are trivial when applied to a specific model is the most significant distinction between the performance of the model-dependent and model-independent modes.

In practice, the most time intensive optimization was *Unify Atomic Propositions*, which is the only routine whose running time is quadratic in the full set of variables. For the three largest benchmarks, this optimization accounted for more than a third of the running time of the compiler.

This system has also been applied to batches of domain specific queries. In one example, a program linked against

Name	Logic	Description	Atomic Propositions
Instantiated over each global variable v			
use-before-define	*CTL*	∃ a path where v is used before it is defined?	D(v), U(v)
never-defined	*CTL*	∃ a path where v is declared but never defined?	L(v), A(v)
never-used	*CTL*	∃ a path where v is defined but never used?	D(v), U(v)
kill-before-use	*CTL*	∃ a path where a definition of v is killed before it is used?	D(v), U(v)
Instantiated over each global pointer variable p			
never-free	*FCTL*	∃ a path where p is malloc'd and never free'd before termination?	M(p), F(p)
illegal-free	*CTL*	∃ a path where p is free'd without having been malloc'd?	M(p), F(p)
double-free	*CTL*	∃ a path where p is free'd twice without being malloc'd in between?	M(p), F(p)
double-malloc	*CTL*	∃ a path where p is malloc'd twice without being free'd in between?	M(p), F(p)
null-dereference	$L\mu$	∃ a path where p is dereferenced after being assigned zero?	Z(p), D(p), R(p)
false-dereference	$L\mu$	∃ a path where p is dereferenced after being tested zero?	D(p), T(p), N(p), R(p)
unneeded-test	$L\mu$	∃ an unneeded test of p being null?	D(p), T(p), N(p), R(p)
Instantiated once per program			
unseeded-random	*CTL*	Is a possibly unseeded random number generated?	rand, srand

D = must define, A = may define, U = may use, L = declaration, M = must malloc, F = may free, R = may dereference, Z = must define zero, T = test zero, N = test non-zero, rand = call to rand(), srand = call to srand(unsigned)

Figure 6: Query Templates used to Generate Test Batches

Benchmark		Unoptimized			Optimized			Statistics		
Code	Queries	Vars	Atomic	Modal	Vars	Atomic	Modal	Time (s)	Passes	Red (%)
188.ammp	251	4382	1589	545	678	97	157	0.9	12	74.4
179.art	163	2702	975	337	682	112	195	0.4	19	47.4
256.bzip2	306	5075	1831	633	1089	200	288	1.1	12	59.1
129.compress	155	2506	902	313	666	128	189	0.7	19	45.2
183.equake	338	5723	2069	713	1095	147	297	1.5	12	62.2
099.go	1146	18379	6607	2297	4174	923	1127	15.6	12	55.8
164.gzip	395	6482	2336	809	1540	261	434	1.6	12	51.4
132.ijpeg	258	4171	1501	521	977	161	268	2.8	19	53.0
124.m88ksim	1659	27454	9901	3425	5045	848	1234	29.9	19	67.4
197.parser	325	5396	1947	673	1342	261	365	2.2	19	51.5
300.twolf	1608	26893	9709	3353	7302	1167	2112	65.1	19	42.6
175.vpr	344	5853	2117	729	1101	172	278	2.0	12	65.8
Totals	6948	115016	41484	14348	25691	4477	6944	123.8	186	56.2

Figure 7: Results of the Optimizing Compiler on the Test Batches

a particular library was checked against a batch of queries designed to ensure that the library was used correctly. The batch was specific to the library, not the program, so there were irrelevant queries pertaining to unused functions in the library. Since the queries spanned a relatively small number of atomic propositions, there was also a noticeable level of redundancy in the sub-expressions. The compiler detected these conditions and the result was a similarly significant reduction in the complexity of the query batch.

5. CONCLUSIONS

This paper has presented an optimizing compiler for batches of temporal logic formulas that is effective at significantly reducing their total complexity. This has been demonstrated by applying the compiler to batches of queries generated from actual benchmark programs that are of the sort used by program verification systems. The reduction in the demand the reduced systems place on a model checking engine has been shown to more than justify the compilation time. This reduction is completely orthogonal to any subsequent reduction achieved by reducing the size of the model. This approach is also applicable to "on the fly" methods [3] that produce a product automaton between the model and the query. The optimizations presented in this paper are only a subset of those that are possible within this framework. The compiler is capable of either working in conjunction with, or in place of, other optimization systems.

Beyond the empirical results, this paper also demonstrated that a compiler-based framework is a rational alternative way to organize a system for optimizing temporal logic formulas. As with any optimizing compiler, this compiler is useful because it transfers responsibility for optimization away from the programmer. This transfer makes it feasible to automatically generate and check large query sets without regard to the specific applicability of each query to the given program. A core set of useful optimizations was reduced to a small number of routines, many operating by simple rules with programming language analogs. The use of a global symbol table allows knowledge of query and component boundaries to be used to speed up the optimization process in ways that are unavailable to optimization systems that only consider raw logical expressions. By using a well-defined intermediate language, commonality can be found among even a heterogeneous collection of formulas. Further, by employing a representation that factors formulas in the same way as an assembly language factors imperative programs, optimizations become permissible that were not possible in the source logic.

Many other approaches to reducing the complexity of temporal logic formulas exist. Some of the most popular involve the construction of Büchi automata [15, 16, 17, 23]. These approaches concentrate on *LTL* optimization, which encompasses only a subset of the full modal mu-calculus. Further, these are again single query optimizations that do not take any special advantage of the redundancy present in query batches over a common domain. Finally, since they produce a specialized type of automata, they are tied to model checking engines that utilize that format. However, in some cases, these approaches are capable of performing optimizations beyond those that have been presented in this paper. For those applications, the analogy with programming language compilers could be extended further by considering equation block systems and Büchi automata as high and low intermediate forms each allowing a disjoint set of optimizations. In this way, the framework presented in the paper could be seen to work with, rather than in place of, these alternate approaches.

The next step in the development of this compiler is to include optimization routines that take advantage of techniques to reduce the complexity of the model used for model checking. As mentioned before, optimizations that reduce the complexity of the model provide performance gains that are orthogonal to the gains made by reducing the query size. However, many model reduction techniques work by performing slicing or dependence analysis on the states of the model. These dependences are driven by the states that are included amongst the atomic propositions of the queries. How batches of queries should be partitioned to produce an optimal set of small models is an open problem. A pass to perform a partitioning based upon query coverage in the model could easily be integrated into this framework.

6. ACKNOWLEDGMENTS

Thanks to Keshav Pingali, David Melski, and Riccardo Pucella for many valuable conversations. Thanks to GrammaTech, Inc. for making their software available and for being supportive of this research.

This work was supported by DARPA contract DAAH01-02-C-R120 and NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-0121401.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. of ICALP 2001, Twenty-Eighth Int. Colloq. on Automata, Languages, and Programming (to appear)*, Crete, Greece, July 2001.

[3] G. Bhat and R. Cleaveland. Efficient model checking via the equational mu-calculus. In *Eleventh Annual Symposium on Logic in Computer Science*, pages 304–312. IEEE Computer Society Press, July 1996.

[4] O. Burkart and B. Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.

[5] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[6] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422, 1992.

[7] R. Cleaveland and B. Steffen. Computing behavioral relations, logically. *ICALP '91, LNCS 510*, 1991.

[8] http://www.grammatech.com/products/codesurfer/.

[9] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[10] E. Emerson and J. Halpern. 'sometimes' and 'not never' revisited: On branching versus linear tune temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[11] E. Emerson and C. Jutla. The complexity of tree automata and logics of programs. *SIAM Journal on Computation*, 29(1):132–158, 1999.

[12] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, Louisiana, 1985.

[13] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings, Symposium on Logic in Computer Science*, pages 267–278, Cambridge, Massachusetts, June 1986. IEEE Computer Society.

[14] J. Ezick, D. W. Richardson, and T. Teitelbaum. Practical model checking and example generation for context-free processes. Technical Report TR2002-1851, Cornell University, August 2001.

[15] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata: Quotienting by simulation equivalences. In *Foundations of Software Technology and Theoretical Computer Science: 22th Conference*, volume 2556 of *Lecture Notes in Computer Science*, pages 157–169, Kanpur, India, 2002.

[16] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Lecture Notes in Computer Science*, volume 2102, pages 53–65. Springer, 2001.

[17] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[18] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

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

[20] W. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[21] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[22] N. Shilov and K. Yi. A new proof of exponential decidability for propositional mu-calculus with program converse. In *III International Conference on Theoretical Aspects of Computer Science*, Novi Sad, Yugoslavia, September 2000.

[23] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Lecture Notes in Computer Science*, volume 1633, pages 247–263. Springer, 2000.

[24] R. S. Streett. Propositional dynamic logic of looping and converse. In *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation*, pages 375–383, Milwaukee, Wisconsin, May 1981.

[25] M. Vardi. Reasoning about the past with two-way automata. In *Lecture Notes in Computer Science*, volume 1443, pages 628–641. Springer, 1998.

[26] M. N. Wegman and K. Zadeck. Constant propagation with conditional branches. *Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.