Using Invariants to Optimize Formal Specifications Before Code Synthesis *

Ralph D. Jeffords and Elizabeth I. Leonard
Naval Research Laboratory (Code 5546)
Washington, DC 20375 USA
{jeffords,leonard}@itd.nrl.navy.mil

Abstract

Formal specifications of required system behavior can be analyzed, verified, and validated, giving high confidence that the specification captures the desired behavior. Transferring this confidence to the system implementation depends on a formal link between requirements and implementation. The automatic generation of provably correct code provides just such a link. While optimization is usually performed on code to achieve efficiency, we propose to optimize the formal specification before generating code, thus providing optimization independent of the particular code generation method. This paper investigates the use of invariants in optimizing code generated from formal specifications in the Software Cost Reduction (SCR) tabular notation. We show that invariants (1) provide the basis for simplifying expressions that otherwise cannot be improved using traditional compiler optimization techniques, and (2) allow detection and elimination of parts of the specification that would lead to unreachable code.

1. Introduction

Formal requirements specifications are useful because they can be analyzed to show that they satisfy critical properties such as safety, security, and timeliness. Additionally, with executable specifications, the user may symbolically execute the system to validate that the specification captures the intended system behavior. Thus, analysis and simulation can provide confidence that a specification is correct. Transferring this confidence to the implementation requires a formal link between requirements and implementation. This formal link may be realized by a sequence of (usually) manual refinements, but the automatic generation of provably correct code provides the highest confidence that the code captures the specified behavior. We have shown in previous work [21] how to construct code from requirements specified in the Software Cost Reduction (SCR) tabular notation. The development of high-quality SCR requirements specifications is supported by a suite of editing and verification tools designed and developed by the Naval Research Laboratory. Automatic code synthesis is consistent with our SCR toolset design philosophy, the goal of which is to automate (as much as possible) the process of system specification, analysis, and implementation using tools and methods designed for practicing engineers.

Both speed and code size are important in code for embedded systems. Compilers generally perform optimizations for speed, while code size optimization is often done by hand on either the source code or the compiled code [28]. Rather than perform optimizations only on the code itself, our approach is to translate the formal specification into an equivalent form that will lead to smaller, and frequently faster, code than the original specification, thus providing optimization independent of the particular code generation method. This will then be followed by more typical optimizations on the code. This paper investigates the use of requirements level invariants in optimizing code generated from executable formal requirements specifications represented in the SCR tabular notation. Invariants are properties that hold in every reachable state of such an executable system. In previous work, we have developed algorithms for automatically generating invariants [16, 17]; these and other invariants that have been established can be used with our techniques.

To illustrate our notion of optimization, we consider a simple state machine Σ with state set $\{x_1, x_2\}$. Associated with Σ is a variable X, whose value represents the current state of Σ, and Boolean variables A and B. The machine Σ changes state based on (and in parallel with) changes in A and B. Here and below, we follow the standard convention in which unprimed variables represent values prior to a

*This research was funded by the Office of Naval Research.

U.S. Government work not protected by U.S. copyright.

transition and primed variables represent values after a transition.

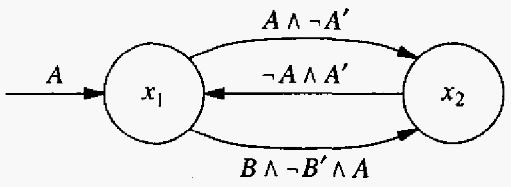

Figure 1. Simple Example State Machine

Figure 1 indicates that, whenever Σ enters state x_1, A holds. This is because (1) in Σ's initial state x_1, A holds, and (2) A holds after the machine enters state x_1 from state x_2; this follows from the label $\neg A \wedge A'$ on the (unique) transition from state x_2 to state x_1 and the convention on primes. Also from Figure 1, if while Σ is in state x_1, A changes from *true* to *false* (i.e., $A \wedge \neg A'$), then the machine exits state x_1. (We assume a semantics that forces a transition whenever one is possible, even if the choice is nondeterministic.) Given this, it is easy to show that A always holds when the machine is in state x_1, or, in other words, the expression $(X = x_1) \Rightarrow A$ is invariant. In fact, a generalization of this observation is the basis for our algorithms for invariant generation [16, 17].

Notice that, in state x_1, if B changes from *true* to *false*, and A holds before this change (i.e., $B \wedge \neg B' \wedge A$ holds), then the machine enters state x_2. However, it is redundant to check that A holds as part of the evaluation $B \wedge \neg B' \wedge A$ since A always holds in state x_1 due to the invariant property given above. Thus, we may simplify the expression labeling this transition to $B \wedge \neg B'$ to avoid checking A.

The transitions illustrated by Figure 1 may be expressed in a tabular format as follows:

source state	guard	target state
x_1	$A \wedge \neg A'$	x_2
x_1	$B \wedge \neg B' \wedge A$	x_2
x_2	$\neg A \wedge A'$	x_1

The SCR method uses similar tables to define state machine transitions. The above table is a compact representation of the function defining X', the value of variable X in the new state. The standard mathematical definition of the function is more complex:

$$X' = \begin{cases} x_2 & \text{if } X = x_1 \wedge ((A \wedge \neg A') \vee (B \wedge \neg B' \wedge A)) \\ x_1 & \text{if } X = x_2 \wedge (\neg A \wedge A') \\ X & \text{otherwise} \end{cases}$$

The simplification performed above can be used to induce a transformation of the table. The cells comprising the guard column of the table are the focus of our simplifications. We are able to perform the simplification because (1) the system is in state x_1 (i.e., $X = x_1$), (2) the invariant $(X = x_1) \Rightarrow A$ always holds, and (3) the previous two facts together imply that A holds. If we take $K = (X = x_1) \wedge (X = x_1 \Rightarrow A)$ to be the *context* of the cell containing the expression $B \wedge \neg B' \wedge A$, then our simplification may be expressed as follows.

> If K is the context for a cell containing the expression E and $K \Rightarrow A$ then, A may be replaced by *true* in E.

A generalization of the above rule to replace any subexpression (rather than just the Boolean variable A) is one of the simplification rules that we have developed. However, the simplification is not yet complete because this rule says that $B \wedge \neg B' \wedge A$ is transformed into $B \wedge \neg B' \wedge$ *true*. Transforming this expression to $B \wedge \neg B'$ is trivial; we simplify using the identity $P \wedge$ *true* $\Leftrightarrow P$. In the general case, we apply this and other standard Boolean simplifications. Transforming the table as described results in a table with a simplified middle row:

source state	guard	target state
x_1	$\neg A'$	x_2
x_1	$B \wedge \neg B'$	x_2
x_2	$\neg A \wedge A'$	x_1

Note that the simplification rule with the same context has also been applied to remove the term A from the guard cell of the first line of the table.

This simple example illustrates that invariants can provide the basis for simplifying expressions that cannot be further simplified without use of those invariants. Such modifications are a form of *contextual simplification*, analogous to contextual rewriting [35], since they involve the use of a context of known facts to aid in the simplification. In addition to the generalization of the above rule, we also develop rules for detecting and eliminating parts of the specification that would lead to unreachable code. Our general approach is to apply a convergent set of contextual simplification rules, each application of which may require additional non-contextual simplification.

While this paper considers only a set of simple rules for simplifying propositional formulas, we are in the process of investigating more sophisticated techniques to include actual algorithms for doing these optimizations, as well as extension to contextual simplification of a more general nature (e.g., simplification of arithmetic expressions).

Section 2 provides background on SCR and on invariants that can be automatically derived from SCR specifications. Section 3 explains how invariants may be used to simplify SCR tables by removing portions of the specification that would lead to unnecessary or dead code. Examples are given to illustrate the utility of invariants in this process. Section 4 discusses related work. Section 5 presents conclusions and ideas for future work.

2. Background

Originally formulated to document the requirements of the flight program of the U.S. Navy's A-7 aircraft [14], the SCR requirements method is designed to support detection and correction of errors during the requirements phase of software development [13, 9]. The SCR toolset provides a user-friendly approach to writing requirements specifications in a tabular format and a number of analysis tools, including a consistency checker [13], a simulator [12], a model checker [10], theorem provers [2, 4], and an invariant generator [16, 17]. By applying the SCR tools to uncover errors, a user can develop high confidence that a specification correctly captures the required system behavior.

The SCR method has been used successfully by many organizations in industry and in government (e.g., Bell Laboratories [15], Grumman [26], Lockheed [7], the Naval Research Laboratory [10, 20], Ontario Hydro [31], and Rockwell Aviation [27]) to develop and analyze specifications of practical systems, including flight control systems [7, 27], weapons systems [10], space systems [6], and cryptographic devices [20]. Most recently, the SCR tools were used, together with a test case generator, by Lockheed Martin to detect a critical error described as the "most likely cause" of a $165 million failure in the software controlling landing procedures in the Mars Polar Lander [5].

2.1 SCR Requirements Model

In SCR the required system behavior is defined in terms of *monitored* and *controlled variables*, which represent quantities in the system environment that the system monitors and controls. The environment nondeterministically produces a sequence of monitored events, where a *monitored event* signals a change in the value of some monitored variable. The system, represented in the model as a state machine, begins execution in some initial state and then responds to each monitored event in turn by changing state. In SCR the system behavior is assumed to be *synchronous*: the system completely processes one set of inputs before processing the next set. Furthermore, the *One Input Assumption* allows at most one monitored variable to change from one state to the next.

To specify the required behavior concisely, the SCR model contains two types of auxiliary variables: *mode classes*, whose values are called *modes*, and *terms*. Each mode is an equivalence class of system states useful in specifying as well as understanding the required system behavior. A term is a state variable defined by an expression over monitored variables, mode classes, or other terms. Mode classes and terms often capture history—the changes that occurred in the values of the monitored variables—and help to make the specification more concise.

The SCR model represents a system as a state machine $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the set of initial states, E^m is the set of monitored events, and T is the transform describing the allowed state transitions [13]. In our model, the transform T is a function that maps a monitored event $e \in E^m$ and the current state $s \in S$ to the next state $s' \in S$. Further, a *state* is a function that maps each *state variable*, i.e., each monitored or controlled variable, mode class, or term, to a type-correct value; a *condition* is a predicate defined on a system state, and an *event* is a predicate requiring that two consecutive system states differ in the value of at least one state variable.

The notation "`@T(c) WHEN d`" denotes a *conditioned event*, which is defined by

$$\texttt{@T(c) WHEN d} \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed conditions c and d are evaluated in the current state and the primed condition c' is evaluated in the next state. The event `@T(c) WHEN d` *occurs* when its defining expression evaluates to *true*. We also define

$$\texttt{@F(c)} \stackrel{\text{def}}{=} @T(\neg c).$$

2.2 The SCR Tables

The transform T is a composition of smaller functions called *table functions*, which are derived from the condition tables, event tables, and mode transition tables in SCR requirements specifications. These tables define the values of the *dependent variables*—the controlled variables, mode classes, and terms. For T to be well-defined, no circular dependencies are allowed in the definitions of the dependent variables. The variables are partially ordered based on the dependencies among the next state values.

Each table defining a term or controlled variable is either a condition table or an event table. A *condition table* associates a mode and a condition in the next state with a variable value in the next state, whereas an *event table* associates a mode and a conditioned event with a variable value in the next state. Each table defining a mode class is a *mode transition table*, which associates a source mode and an event with a target mode. Our formal model requires the information in each table to satisfy certain properties, guaranteeing that each table describes a total function [13]. Some SCR tables may be modeless, i.e., they define the value of a variable without referring to any mode class.

Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

Table 1. Mode Transition Table for `Pressure`**.**

Mode	Conditions	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Safety_Injection	Off	On

Table 2. Condition Table for `Safety_Injection`**.**

To illustrate the SCR tabular notation, three example tables are presented. These tables define the values of the three dependent variables in a simplified version of a safety injection system (SIS) [13] for a nuclear power plant. The SIS system monitors water pressure, and if the pressure is too low, the system injects coolant into the reactor core.

Table 1 is a mode transition table defining the new value of the mode class Pressure as a function of the current mode and the monitored variables. For example, the first row of the table states that if the current mode is TooLow and the water pressure becomes greater than or equal to the Low threshold, the new mode is Permitted.

Table 2 is a condition table defining the value of the controlled variable Safety_Injection as a function of the modes and the term variable Overridden. The first row states that in the High or Permitted modes, Safety_Injection is Off. The second row states that in the mode TooLow, if Overridden is *true* then Safety_Injection is Off, and if Overridden is *false* then Safety_Injection is On.

Table 3 is an event table defining the term Overridden as a function of the current mode and the monitored variables. The first row describes the behavior when the mode of the system (i.e., the value of Pressure in the old state) is either TooLow or Permitted. In either of these modes, if Block switches to On when Reset is Off, then the new value of Overridden is *true*, but if the Pressure becomes High or Reset switches to On, then the new value of Overridden is *false*.

2.3 Invariants and Code Generation

We consider two forms of invariants in SCR: *state invariants*, expressions over a single state that hold in each reachable state of the system, and *transition invariants*, expressions over two states that hold for each reachable pair of consecutive states. We have designed two algorithms [16, 17] for constructing state invariants from the tables defining the dependent variables in an SCR specification. Suppose that dependent variable r has values in a finite set $\{v_1, v_2, ..., v_n\}$. If the value of r is defined by a mode transition table or an event table, then, for each v_i, the algorithms generate invariants of the form

$$r = v_i \Rightarrow C_i,$$

where C_i is a predicate over the variables in Σ on which r depends. Invariant generation from SCR tables is based on the following idea: In an SCR specification, $r = v_i \Rightarrow C_i$ is an invariant if 1) C_i is always true when r's value changes to v_i, and 2) an event falsifying C_i unconditionally causes r to have a value other than v_i. Since stronger invariants may be computed with knowledge of previously computed invariants, the full algorithms repeat the computations of the invariants until a fixpoint is reached. The current implementation of the SCR invariant generator applies our algorithms to both mode transition tables and event tables. State invariants constructed from a mode transition table are called *mode invariants*.

We have also developed two prototype code synthesizers that construct C source code from an SCR requirements specification [21]. The two synthesizers, each using a different code generation strategy, are based on Paige's APTS program transformation system [30]. The first strategy uses rewrite rules to transform the parse tree of an SCR specification into a parse tree for the corresponding C code. The second strategy associates a relation with each node of the specification parse tree. Each member of this relation acts as an attribute, holding the C code corresponding to the tree at the associated node; the root of the tree has the entire C program as its member of the relation. The generated code is efficient but has not been optimized.

Mode Pressure	Events	
TooLow, Permitted	@T(Block = On) WHEN Reset = Off	@T(Pressure = High) OR @T(Reset = On)
High	False	@F(Pressure = High)
Overridden	True	False

Table 3. Event Table for `Overridden`**.**

$$\texttt{Overridden}' = \begin{cases} \texttt{true} & \text{if @T(Block=on) WHEN Reset=off} \\ & \quad \text{AND Pressure in \{TooLow, Permitted\}} \\ \texttt{false} & \text{if @T(Pressure=High) OR @T(Reset=on) WHEN} \\ & \quad \text{Pressure in \{TooLow, Permitted\}} \\ & \quad \text{OR } \textit{@F(Pressure= High) WHEN Pressure=High} \\ \texttt{Overridden} & \text{otherwise} \end{cases}$$

Figure 2. Functional Definition of Overridden Event Table.

3. Simplifying SCR Tables Using Invariants

This section presents two simplification rules that make use of invariants: (1) a rule to remove unreachable parts of the specification and (2) another rule to remove redundant parts of the specification. Since invariants are properties that hold in any reachable state, invariants may be used to simplify the expression of the next state function, the function from which code is ultimately generated. Note that, to simplify an expression E, it is not sufficient to simply conjoin the invariants with E and apply some simplification procedure, because this might entail the simplification of *both* E and the invariants, when all we want to simplify is E itself. Thus, some form of expression simplification that uses the invariants as context is desired.

3.1. Contexts

For each cell to be simplified, several different forms of information may be assumed as context: the current value of the associated mode class, a constraint on the old value of the variable being defined in the table, and the set of invariants. However, for a technical reason (as explained in the appendix) the contextual information involving the old value of the variable may only be used as context for the Rule Remove-Unreachable.

A. THE MODE CLASS (Both Rules): Usually an event table in SCR has an associated mode class M; that is, the value of the variable defined by the table is described as a function of that mode class and an event. Except for mode-less event tables, the mode in the old state can be used as part of a cell's context. For mode transition tables, the value of the mode in the old state can be used as context for the cell in the corresponding event column. For example, in Table 3 the mode context for the cell "@F(`Pressure = High`)" obtained from the associated mode class Pressure is "`Pressure = High`," while the mode context for the cell in row 2 in Table 4 on page 7 is "`CruiseMode = Inactive`."

B. CONSTRAINT ON THE OLD VALUE (Rule Remove-Unreachable Only): For an event table, a constraint on the old value of the variable being defined can also be used as part of the context of a cell. Event tables have a default "no change" condition, meaning that for a given cell, we only need to consider the value of the variable if the actual value of the variable changes. This is supported by the following property related to the formal definition of tables as given in [13], of which Figure 2 is an example.

Property 1 *For a variable r having the set of possible values $\{v_1, \ldots, v_n\}$, the function definition*

$$r' = \begin{cases} v_1 & \textit{if } P_1 \\ \ldots & \ldots \\ v_n & \textit{if } P_n \\ r & \textit{otherwise} \end{cases}$$

is equivalent to the definition

$$r' = \begin{cases} v_1 & \textit{if } P_1 \wedge r \neq v_1 \\ \ldots & \ldots \\ v_n & \textit{if } P_n \wedge r \neq v_n \\ r & \textit{otherwise} \end{cases}$$

if the set $\{P_1, P_2, ..., P_n\}$ satisfies Disjointness, i.e. $i \neq j \Rightarrow \neg(P_i \wedge P_j)$ for all $1 \leq i, j \leq n$.

This property also holds when only conjoining $r \neq v_i$ for some subset of the P_i rather than all of the P_i. Thus for each cell in the definition of the new value r' defined by an event table we have the context $r \neq v$ where v is the value below the double line at the bottom of the column containing that designated cell. For example, in Table 3 this gives the context for the "@F(`Pressure` = `High`)" cell as "`Overridden` $\neq$ `false`."

C. THE INVARIANTS (Both Rules): Though any state invariant of Σ can be used as context, this paper only considers mode invariants, i.e., state invariants of the form $M = m_i \Rightarrow Q_i$, where M is a mode class name and Q_i is a predicate defined on state variables of Σ.

3.2. Simplification Rules

For an intuitive presentation of our simplification techniques using invariants, we express the simplifications in terms of transformations of the cells of an SCR table. A tool implementing these simplifications would define these transformations directly in terms of the conditional expressions defining the semantics of each table, but the results would be equivalent. For example, consider the event table in Table 3. This table, which is adapted from the SCR specification of a safety injection system [13], describes how the value of the variable `Overridden` is updated. The semantics of Table 3 is given as the conditional expression of Figure 2.

Our simplifications apply to cells containing the event expressions occurring in event tables (e.g. the cells above the double line with header "Events" in Table 3) and mode transition tables (the cells with the header "Event" in Table 4). As a special case a cell may contain $false$, meaning that the case is impossible. Our simplifications are *contextual* in the sense that we shall simplify cells in the context of the given invariants plus additional facts as described above. In this paper, we present only two rules, both defined over a logical expression K, the context of a cell, and E, the event expression contained in that cell.

Context for Remove-Redundancy: $K = (M = m) \wedge I$, where (a) m is the old value of the mode class M associated with the cell; (b) I is some state invariant (in the old state).

Rule Remove-Redundancy: If E is an expression containing a subexpression Q for a cell associated with mode value m, and $K \Rightarrow Q$ is a tautology, then E may be simplified by replacing each occurrence of Q within E with $true$.

Intuitively, this rule says that if cell E is being evaluated in a context where both K and Q are true, then effectively the value of E is unchanged by treating each occurrence of Q as $true$. If applying this rule simplifies E, one would naturally further simplify E using standard simplification algorithms. In this paper, we shall only apply Remove-Redundancy to mode transition tables.

Context for Remove-Unreachable: $K = (M = m) \wedge I \wedge (r \neq v)$, where (a) m is the old value of the mode class M associated with the cell containing E, (b) I is some state invariant (in the old state), and (c) v is the new value of r associated with the cell.

Rule Remove-Unreachable: If $K \wedge E \Rightarrow false$ is a tautology, then E may be replaced by $false$.

Obviously, if the context is $false$, then the transition associated with this cell will never occur. Replacing the cell entry with $false$ results in a clearer and more concise specification.

Next, we illustrate several simplifications using Rule Remove-Redundancy. Table 4 shows the mode transition table for a Cruise Control system [11]. Applying our previously developed invariant generation algorithms, produces the following two invariants for the cruise control specification: (1) `CruiseMode` = `Inactive` $\Rightarrow$ `IgnOn` and (2) `CruiseMode` = `Override` $\Rightarrow$ `IgnOn`$\wedge$`EngRunning`. Consider Row 3 of Table 4 and let E be the event expression from this row. Let I be the invariant (1) and take the context K to be I together with the mode context for this row, `CruiseMode` = `Inactive`. Together these two parts of the context imply `IgnOn`. Applying Rule Remove-Redundancy with Q = `IgnOn` eliminates "`And IgnOn`" from the end of the event expression in the cell (marked in italics). Code generated from the simplified table will be smaller and faster than code generated from the original table. Similarly, we can simplify line 9 of the mode transition table using invariants (1) and (2) to remove the expression "`And IgnOn And EngRunning`" (shown in italics).

Finally, we illustrate how applying Rule Remove-Unreachable will lead to elimination of a row of Table 3. This corresponds to elimination of a part of the specification that would produce dead code during synthesis. Let E be the cell containing @F(`Pressure` = `High`) in the event table given in Table 3 and let I be `Pressure = High` $\Rightarrow$ `Overridden = false`, one of the generated state invariants for this system. Let the context K be the invariant I together with the mode class information, `Pressure=High`, and the old state value information, `Overridden` $\neq$ `false`. The three constraints of the context K taken together simplify to $false$; and thus by the Rule Remove-Unreachable the cell itself can be replaced by $false$. Because all the cells in the second row of the table now are $false$, the entire row of the table can be eliminated.

Old Mode	Event	New Mode
1 `Off`	@T(`IgnOn`)	`Inactive`
2 `Inactive`	@F(`IgnOn`)	`Off`
3 `Inactive`	@T(`Lever = const`) WHEN `EngRunning` AND NOT `Brake` *AND IgnOn*	`Cruise`
4 `Cruise`	@F(`IgnOn`)	`Off`
5 `Cruise`	@F(`EngRunning`)	`Inactive`
6 `Cruise`	@T(`Brake`) OR @T(`Lever = off`)	`Override`
7 `Override`	@F(`IgnOn`)	`Off`
8 `Override`	@F(`EngRunning`)	`Inactive`
9 `Override`	@T(`Lever = resume`) OR @T(`Lever = const`) WHEN NOT `Brake` *AND IgnOn AND EngRunning*	`Cruise`

Table 4. Mode Transition Table for Mode Class Variable CruiseMode.

The more compact table is shown in Table 5. The new table will produce less code during synthesis because it omits the part of the table that would lead to the construction of dead code.

There is one special case of Remove-Unreachable that bears mention. If there is an invariant of the form $\mathtt{M} = \mathtt{m} \Rightarrow \mathit{false}$, any row of a table having $\mathtt{M} = \mathtt{m}$ as the mode class context can be eliminated from the table. This one-step optimization is equivalent to a series of applications of Remove-Unreachable (one for each cell in the row), resulting in a row of cells having the value false, followed by the elimination of the row.

4. Related Work

The language LUSTRE [8], developed at VERIMAG, is conceptually similar to the SCR language: it provides a deterministic language, in which all non-input variables are simultaneously updated in response to some change in the input environment. Efficient code generation is an integral part of the LUSTRE toolset, and is based on the use of a "control automaton" that remembers a limited part of the old state of the system. The VERIMAG group has also extended LUSTRE into the hardware area by adding syntactic sugar for array structures and circuit layout information, which the Pollux tool uses to automatically configure the hardware gates in Programmable Active Memory [34].

Early work on logical simplification in the 1950's and 1960's addressed Boolean minimization with respect to some measure (such as fewest number of literals in sum-of-products form) resulting in the well-known Quine-McCluskey method [33, 25]. Later developments extended simplification over first-order theories with interpreted symbols: Loveland and Shostak [24] extended Quine's method of prime implicants, while Zhang [36] gives a general framework for simplification via *contextual rewriting*, i.e., rewriting formulae in the context of additional information. This latter work has been extended to consider use of decision procedures in manipulating the context during rewriting [3]. The most sophisticated of these techniques have resulted in implementations of powerful theorem provers, e.g., SIMPLIFY, which is based on the work of Nelson [29]. The two rules we have given are special cases of contextual rewriting as originally defined by Remy [35], who first coined the terminology "contextual rewriting."

Complementing the early work on logic simplification in the 1950's and 1960's was the development of techniques for machine simplification, e.g., the minimization of the number of states of incompletely specified finite state machines [32]. The monograph by Kam et al. gives a modern perspective on this subject [18].

Invariants have been used for optimization during code generation for many years, but for the most part such invariants are related to implementation details rather than requirements level invariants of reactive, embedded systems that we generate from SCR specifications. For example, "loop invariants" about the relative values of variables in a loop are used during the classic strength-reduction compiler optimization technique [1] and the finite differencing program transformation technique [30]. More recent work on strengthening such invariants has led to additional optimization as well as providing a more general approach called incrementalization [23]. Another application of invariants during code generation, but at a higher level akin to requirements, is the technique of run-time code generation [22, 19]. In this method, specialized code is generated at run-time, given invariants based upon the known input values for a specialized (often one-time) use of a program.

In our simplification of the cells of a table, we use the old state value of the variable as means of restricting the calculation of the variable's new value to only the cases where there is to be a change from the old value of the variable. This check of the old value of the variable could also be generated as part of the synthesized code. If the check were

Mode	Events	
TooLow, Permitted	@T(Block = On) WHEN Reset = Off	@T(Pressure = High) OR @T(Reset = On)
Overridden	True	False

Table 5. Simplified Event Table for `Overridden`**.**

$$\texttt{Overridden}' = \begin{cases} \texttt{true} & \text{if } @\texttt{T(Block=on)} \text{ WHEN } \texttt{Reset=off} \\ & \quad \text{AND } \texttt{Pressure} \text{ in } \{\texttt{TooLow, Permitted}\} \\ \texttt{false} & \text{if } @\texttt{T(Pressure=High)} \text{ OR } @\texttt{T(Reset=on)} \text{ WHEN} \\ & \quad \texttt{Pressure} \text{ in } \{\texttt{TooLow, Permitted}\} \\ \texttt{Overridden} & \text{otherwise} \end{cases}$$

Figure 3. Functional Definition of Simplified Overridden Event Table.

generated such that it was a preliminary check before the rest of the calculations were performed, it would optimize the code by preventing unnecessary calculations. This sort of incremental update to the variable (i.e., basing its new value upon its old value) as well as the LUSTRE control automaton approach to compilation are similar to finite differencing [30].

5. Conclusions and Future Work

Though at a preliminary stage, the work reported in this paper shows that some benefit can be derived from using invariants to simplify SCR specifications. In future work, we plan to implement a tool that applies more general invariants (to include transition invariants) to simplify SCR tables using algorithms that support contextual simplification in the more general setting of interpreted first-order theories, (e.g., arithmetic expressions, enumeration expressions, etc.). While the simple idea of a cell and its context provide an intuitive framework for explaining the optimization of SCR specifications, the implementation will perform the optimizations directly on the underlying functional definitions. The output of this tool will then be used as input for our previously developed code synthesizers, allowing us to produce code that has been optimized. We plan to perform experiments to determine the amount of improvement the optimizations provide for typical SCR specifications. We also plan to implement the finite differencing optimization described in Section 4.

Acknowledgments. We thank our colleague Connie Heitmeyer for the "constraint on the old value" form of context, and suggestions for improving the presentation of this work. The anonymous reviewers and our colleagues Myla Archer and Michael Colón also provided helpful comments. Myla helped to improve the proof in the appendix and she furnished Figure 1.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, 1988.

[2] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4), February 2001.

[3] A. Armando and S. Ranise. Constraint contextual rewriting. *J. of Symbolic Computation*, 36(1/2):193–216, July/August 2003.

[4] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '2000)*, Berlin, Mar. 2000.

[5] M. Blackburn, R. Knickerbocker, and R. Kasuda. Applying the test automation framework to the Mars lander touch-down monitor. In *Lockheed Martin Joint Symposium*, 2001.

[6] S. Easterbrook, R. Lutz, R. Covington, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1), Jan. 1998.

[7] S. R. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)*, Gaithersburg, MD, June 1994.

[8] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Softw. Engin.*, 18(9):785–793, Sept. 1992.

[9] C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., New York, NY, second edition, 2002.

[10] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), Nov. 1998.

[11] C. Heitmeyer, J. Kirby, Jr., and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[12] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

[13] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[14] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, NRL, Wash., DC, 1978.

[15] S. D. Hester, D. L. Parnas, and D. F. Utter. Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977, Oct. 1981.

[16] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, Nov. 1998.

[17] R. D. Jeffords and C. L. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering*, Aug. 2001.

[18] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer, Boston,MA, 1996.

[19] S. Kamin. Routine run-time code generation. In *Proc. OOPSLA'03*, pages 208–220, Anaheim,CA, Oct. 2003.

[20] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Computer Society Press, Dec. 1999.

[21] E. I. Leonard and C. L. Heitmeyer. Program synthesis from formal requirements specifications using APTS. *Higher-Order and Symbolic Computation*, 16(1/2):63–92, Mar/June 2003.

[22] M. Leone and P. Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, Feb. 1996.

[23] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Strengthening invariants for efficient computation. In *Proc. 23rd Annual ACM Symp. on Principles of Prog. Lang. (POPL)*, St. Petersburg Beach, FL, Jan. 1996.

[24] D. W. Loveland and R. E. Shostak. Simplifying interpreted formulas. In W. Bibel and R. Kowalski, editors, *Proc. 5th Conf. on Automated Deduction (CADE)*, volume 87 of *LNCS*, pages 97–109, Les Arcs, France, 1980. Springer-Verlag.

[25] E. J. McCluskey. Minimization of boolean functions. *Bell Sys. Tech. J.*, 35:1417–1444, Nov. 1956.

[26] S. Meyers and S. White. Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY, July 1983.

[27] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd ACM Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[28] M. Naik and J. Palsberg. Compiling with code-size constraints. *ACM Transactions on Embedded Computing Systems*, 3(1):163–181, February 2004.

[29] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford Univ., Stanford, CA, June 1981.

[30] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[31] D. L. Parnas, G. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2), April–June 1991.

[32] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Trans. on Electronic Computers*, EC-8:356–367, Sept. 1959.

[33] W. V. Quine. The problem of simplifying truth functions. *Am. Math. Monthly*, 59:521–531, Oct. 1952.

[34] F. Rocheteau and N. Halbwachs. POLLUX: a LUSTRE based hardware design environment. In P. Quinton and Y. Robert, editors, *Conf. on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas,France, 1991.

[35] H. Zhang. Implementing contextual rewriting. In M. Rusinowitch and J.-L. Remy, editors, *Proc. Third Int'l Workshop on Conditional Term Rewriting Systems (CTRS)*, volume 656 of *LNCS*, pages 363–377. Springer, 1992.

[36] H. Zhang. Contextual rewriting in automated reasoning. *Fundamenta Informaticae*, 24(1/2):107–123, Sept. 1995.

A. Soundness

We have formally proved that both rules are sound, each with its own particular definition of context. Although these rules appear to be quite simple, careful attention to the allowable context is required. It would seem intuitive that the constraint on the old value of the variable v could be used as context with Rule Remove-Redundancy since a transformation via Property 1 preserves the function. But this is unsound: it is easy to find an SCR table for which application of Rule Remove-Redundancy with the constraint $r \neq v$ as part of the context introduces nondeterminism.

We now present the proof of the soundness of Rule Remove-Unreachable for an event table. The proof is for the more general case of application of the rule to all cells simultaneously. To avoid clutter we suppress explicit mention of the state. We consider the semantics of an event table defining the new value of a state variable r as a conditional expression:

$$F_r = \begin{cases} v_1 & \text{if } G_1 \\ \ldots & \ldots \\ v_n & \text{if } G_n \\ r & \text{otherwise} \end{cases}$$

where the set of guards $G_i \overset{def}{=} (M = m_i) \wedge E_i$, $i = 1, \ldots, n$ are mutually disjoint; this ensures that this conditional form represents a function.

For every i, the Remove-Unreachable context for E_i is

$$K_i = (M = m_i) \wedge I_i \wedge (r \neq v_i)$$

where I_i is some state invariant, which may be chosen differently for each i. Recall:

> **Rule Remove-Unreachable**: If $(K_i \wedge E_i) \Rightarrow \mathit{false}$ is a tautology, then replace E_i with false.

After applying this rule for every i, we have a new definition F_r^*, with each G_i in the definition of F_r replaced by G_i^*, where

$$G_i^* = (M = m_i) \wedge E_i^*, \qquad (1)$$

in which

$$E_i^* = \begin{cases} \mathit{false} & \text{if } (K_i \wedge E_i) \Rightarrow \mathit{false} \\ E_i & \text{otherwise.} \end{cases} \qquad (2)$$

Note that (1) and (2) together imply that $G_i^* \Rightarrow G_i$.

For F_r^* to define a well-formed table function, the G_i^* must be mutually disjoint. But this fact is easily established since the only modification to F_r is to (possibly) replace some of the E_i by false.

Theorem 1 *Semantically, with respect to the reachable states of the system, $F_r \equiv F_r^*$.*

Proof: In our proof, we may assume that all evaluation takes place in a reachable state.

The definition of F_r^* expands to

$$r' = \begin{cases} v_1 & \text{if } G_1^* \\ \ldots & \ldots \\ v_n & \text{if } G_n^* \\ r & \text{otherwise} \end{cases} \qquad (3)$$

and the definition of F_r expands to

$$r' = \begin{cases} v_1 & \text{if } G_1 \\ \ldots & \ldots \\ v_n & \text{if } G_n \\ r & \text{otherwise.} \end{cases} \qquad (4)$$

We must show that the values of these two case expressions are equal. We need only consider two cases.

CASE [$\forall i : \neg G_i^*$]: In this case, the conditional expression in (3) evaluates to r. Using Property 1 on page 5 we can rewrite the conditional expression in (4) to:

$$\begin{cases} v_1 & \text{if } G_1 \wedge (r \neq v_1) \\ \ldots & \ldots \\ v_n & \text{if } G_n \wedge (r \neq v_n) \\ r & \text{otherwise.} \end{cases} \qquad (5)$$

To show that this expression also evaluates to r, it suffices to show

$$\forall i : \neg(G_i \wedge (r \neq v_i)) \qquad (6)$$

holds. But if (6) is false then there is some i such that $G_i \wedge (r \neq v_i)$ holds. In this case, $r \neq v_i$, and since G_i holds, we also have $M = m_i$ and E_i. Further, I_i holds because state invariants hold in any reachable state. Therefore, we know that $K_i = (M = m_i) \wedge I_i \wedge (r \neq v_i)$ holds. Thus, $K_i \wedge E_i$ holds, which means that $K_i \wedge E_i \Rightarrow \mathit{false}$ does *not* hold. ¿From this, we know $E_i^* = E_i$ (by (2)), and hence, $G_i^* = G_i$ (by (1)). Because G_i holds, G_i^* also holds. But this contradicts the assumption $\forall i : \neg G_i^*$. Therefore, we have established (6).

CASE [$\exists i : G_i^*$]: Choose i such that G_i^* holds. Then the case expression in (4) evaluates to v_i. Since $G_i^* \Rightarrow G_i$, G_i also holds. But this means that the value of the case expression in (3) also evaluates to v_i. Hence, the values of the two case expressions are equal. ■