Generating Optimized Code from SCR Specifications *

Tom Rothamel †
Yanhong A. Liu
Stony Brook University
{rothamel, liu}@cs.sunysb.edu

Constance L. Heitmeyer
Elizabeth I. Leonard
Naval Research Laboratory
{heitmeyer,leonard}@itd.nrl.navy.mil

Abstract

A promising trend in software development is the increasing adoption of model-driven design. In this approach, a developer first constructs an abstract model of the required program behavior in a language, such as Statecharts or Stateflow, and then uses a code generator to automatically transform the model into an executable program. This approach has many advantages—typically, a model is not only more concise than code and hence more understandable, it is also more amenable to mechanized analysis. Moreover, automatic generation of code from a model usually produces code with fewer errors than hand-crafted code.

One serious problem, however, is that a code generator may produce inefficient code. To address this problem, this paper describes a method for generating efficient code from SCR (Software Cost Reduction) specifications. While the SCR tabular notation and tools have been used successfully to specify, simulate, and verify numerous embedded systems, until now SCR has lacked an automated method for generating optimized code. This paper describes an efficient method for automatic code generation from SCR specifications, together with an implementation and an experimental evaluation. The method first synthesizes an execution-flow graph from the specification, then applies three optimizations to the graph, namely, input slicing, simplification, and output slicing, and then automatically generates code from the optimized graph. Experiments on seven benchmarks demonstrate that the method produces significant performance improvements in code generated from large specifications. Moreover, code generation is relatively fast, and the code produced is relatively compact.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—tools; D.3.4 [*Programming Languages*]: Processors—code generation, optimization

General Terms Performance, Languages

Keywords code generation, code synthesis, SCR, requirements specifications, formal specifications, optimization

* The research of Tom Rothamel and Yanhong Liu is supported by ONR grant N000140410722. The research of Constance Heitmeyer and Elizabeth Leonard is supported by the Office of Naval Research.

† Research performed at NRL.

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

1. Introduction

Originally formulated to document the requirements of the flight program of the U.S. Navy's A-7 aircraft [14], SCR (Software Cost Reduction) is a tabular notation for specifying the required behavior of software systems. The SCR notation, which has an explicit state machine semantics [13], has been used by many organizations in industry and in government (for example, Lockheed [7], the Naval Research Laboratory [11, 18], and Ontario Hydro [23]) to specify and analyze the requirements of practical systems, including flight control systems [7, 22], weapons systems [11], space systems [5], and cryptographic devices [18]. The SCR toolset [10] provides a user-friendly approach to writing requirements specifications and a suite of analysis tools for analyzing them. The toolset includes a consistency checker [13], a simulator [12], a model checker [11], theorem provers [2, 4], and an invariant generator [15, 17].

One major advantage of writing specifications in a language such as SCR is that the developer can verify automatically that critical properties hold and can validate using simulation that the specification captures the intended behavior. This provides a high degree of confidence in the correctness of the specification. One major advantage of code automatically generated from specifications is that the code usually contains fewer errors than hand-crafted code.

Missing from the current SCR toolset, however, is a tool that can generate optimized programs from SCR specifications. This paper describes a practical method and tool which transform SCR specifications into efficient source code. Optimization is a necessary part of code generation from specifications. Because the goal of specification writers is to produce a clear and concise specification, efficiency of the implementation is rightfully not a major concern. To make the resulting implementation efficient, a code generator needs to include optimization. This is especially important in generating code for embedded systems, which may be limited in both processor power and memory.

This paper describes a method that transforms an SCR specification into an execution-flow graph, a representation suitable for optimization, and then introduces a systematic method for creating the graph, for optimizing it, and for generating code from the result. Our techniques, which have been implemented in a tool called OSCR (Optimizer for SCR), are fully automatic, generate code efficiently, and produce code that is efficient and relatively compact. The paper is organized as follows: After Section 2 reviews SCR, Section 3 introduces the execution-flow graph representation of an SCR specification and our optimization method. Section 4 presents a method for synthesizing an execution-flow graph from a specification, and Sections 5–7 describe three optimization techniques—namely, input slicing, simplification, and output slicing. Section 8 describes how code is generated from the execution-flow graph. Section 9 describes the OSCR tool, and presents an experimental evaluation of our method. Finally, Section 10 discusses related and future work, and presents some conclusions.

2. SCR Specifications

In SCR, the required system behavior is defined as a relation on *monitored* and *controlled variables*, which represent quantities in the 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 defined by an SCR specification begins execution in some initial state and then changes state in response to monitored events. The SCR semantics [13] make two basic assumptions. The *One Input Assumption* allows at most one monitored variable to change from one state to the next, while the *Synchrony Assumption* requires that no new monitored event arrive until the system has processed the current monitored event.

An SCR specification usually contains two types of auxiliary variables: *mode classes*, whose values are called *modes*, and *terms*. Each mode defines an equivalence class of system states useful both in specifying and in understanding the required system behavior. A term is a state variable defined in terms of monitored variables, mode classes, and possibly other terms. Mode classes and terms are used to capture the system's input history—the changes that occurred in the values of the monitored variables—and help make the specification more concise.

In the SCR model [13], a system is represented 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 $T : E^m \times S \rightarrow S$ is the transform describing the allowed state transitions. 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

$$\texttt{@T(c) WHEN d} \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d, \qquad (1)$$

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 state in which c is evaluated is called the *prestate* and the state in which c' is evaluated the *poststate*. In this definition, $\neg c \wedge c' \wedge d$ is an example of a two-state expression because it is evaluated in both the prestate and the poststate, while $\neg c \wedge d$ is an example of a one-state expression because it is evaluated in a single state.

The SCR Tables. To compute the next state, the transform function T composes the *table functions* derived from the condition tables, event tables, and mode transition tables in SCR specifications [13]. These tables define the values of the *dependent variables*—the controlled variables, mode classes, and terms in the specification. 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 their dependencies in the new state.

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

Proper SCR Style. To be well-formed, an SCR specification must satisfy the definitions and properties above. Another important property of an SCR specification is that it use modes effectively.

current mode **setting**	event	new mode **setting'**
`disabled`	@T(`switch_on`)	`enabled`
`enabled`	@T(`not switch_on`)	`disabled`

Table 1. The mode transition table defining `setting`.

	`sensed` $>=$ `target`	`sensed` $<$ `target`
desired =	False	True

Table 2. The condition table defining `desired`.

`setting` $=$ `disabled`	True	False
`setting` $=$ `enabled`	not `desired`	`desired`
heat =	False	True

Table 3. The condition table defining `heat`.

High-quality SCR specifications use modes to partition the system states into equivalence classes. In a given mode, the system will react differently to a given input or sequence of inputs than it will react in a different mode. While our method produces correct code for any well-formed SCR specification, our optimizations work best on SCR specifications that use modes effectively and that define a modest number of modes and a small number of mode classes.

Running Example. To demonstrate our method, we present a simple SCR specification of a thermostat. This specification has three monitored variables and three dependent variables. The monitored variables are `target`, the target temperature set by the user; `sensed`, the temperature sensed by the environment; and `switch_on`, which is true when the user turns on temperature control and false otherwise. Tables 1–3 define the three dependent variables in this specification, `setting`, `desired`, and `heat`. Table 1 is a mode transition table defining the mode class `setting`, which determines whether the system is controlling the temperature (`setting = enabled`) or not (`setting = disabled`). Table 2 is a modeless condition table defining a term `desired` which is true when heating is desired (i.e., the sensed temperature is less than the target temperature set by the user) and false otherwise. Table 3, a moded condition table, defines the controlled variable `heat`, setting it to true when `setting = enabled` and `desired` are true, and to false otherwise.

This specification has been designed to demonstrate our optimizations. It is stylistically proper SCR, as the mode class `setting` changes only when the user changes `switch_on`, a rare occurrence. At the same time, it is not the minimal specification that we could write; for example, the variable `desired` could be eliminated and its use in defining `heat` could be replaced by the conditions defining `desired`. The reason we use two variables is to demonstrate all three optimization techniques.

Each SCR table consists of a number of cells, each containing a logical expression. Prior to code generation, each table is translated into an internal form, which represents the information in each cell as the conjunction of a logical expression and an assignment. Both the logical expression and the assignment are two-state SCR expressions, with logical expressions in cells extracted from condition tables represented in the poststate. The logical expression representing each cell qualified by a mode is the conjunction of the mode (e.g., `setting=enabled`) with the expression in the cell. SCR events are represented in the form shown in (1). This transformation allows us to treat one- and two-state expressions in the same manner.

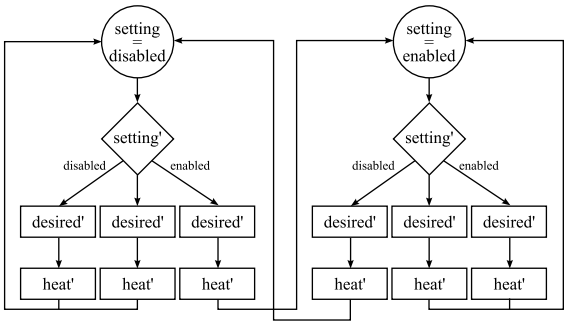

Figure 1. The initial execution-flow graph created for the thermostat specification. Circles represent header nodes, diamonds switch nodes, and rectangles table nodes. Edges leaving switch nodes are labeled with the assignment that occurs, if any.

3. The Execution-Flow Graph

The key to producing optimized code from an SCR specification is to express the transform function in many forms, where each form is specialized for some subset of possible transitions. If only a single form was used for all transitions, we could only exploit redundancy found in the single transform function. While such an approach could lead to small improvements in efficiency (as suggested in [16]), producing different forms from the transform function allows for much larger improvements. By creating a representation of the transform function specialized for each kind of transition, we trade an increase in code size for a decrease in the time needed to evaluate the transform function.

To produce specialized representations of the transform function, the tabular representation of an SCR specification is no longer sufficient. Specialized representations are needed that describe 1) the mode the system is in during the prestate, 2) the mode the system will enter in the poststate, and 3) the input that caused the transition to occur. A representation of an SCR specification suitable for generating optimized code is an *execution-flow graph*, a control-flow graph that explicitly represents the modes the system is in while awaiting input, and the calculations that take place when a state transition occurs. This graph describes an abstract system that pauses execution to wait for input, rather than the event-driven system that will eventually be implemented. The graph contains three types of nodes: **header nodes**, which represent program states in which the system is waiting for input; **switch nodes**, which represent the computation of new values of mode classes; and **table nodes**, which represent the computation of new values of other dependent variables. Edges in the graph indicate control flow between these nodes, with the choice of which edge is traversed determined by the node that is exited. As an example, Figure 1 shows the execution-flow graph we initially construct for the thermostat specification. The three classes of nodes are described in more detail below:

- A **header node** represents the set of all program states in which the program is waiting for input. Each header node is associated with a set of modes, one for each mode class. There is a single unique header node for each set of modes. This header node represents all states in which the program is in the set of modes associated with that header node. If the specification does not contain any mode classes, the execution-flow graph created from that specification will have only a single header node, representing all states of the SCR state machine. Header nodes have either a single outgoing edge that can be used for all inputs, or one outgoing edge for each monitored variable, allowing the transform function to be specialized for that input. Header nodes are the only node class with multiple incoming edges.
- A **switch node** represents the computation of the value of a mode class during a transition. These nodes, which allow control to move along a branch based on the computed value of the mode class, are necessary to ensure that control reaches a header node corresponding to the newly computed mode. Thus switch nodes allow us to encode mode transitions into the execution-flow graph.

  A switch node represents the computation of a single variable value. Each switch node contains one or more branches, with each branch labeled by a logical expression and an assignment. When an expression evaluates to true, then the assignment is performed. When no expression evaluates to true, no assignment occurs. The logical expressions associated with different branches of a switch node must be disjoint, as this is a property of well-formed SCR specifications [13]. Each switch node has one outgoing edge for each branch, and one outgoing edge used when no logical expression is valid. If an expression evaluates to true, control is transferred from the switch node along the edge associated with that expression.
- **Table nodes** represent all other computations. Each table node is associated with a number of cells. Each cell contains a two-state expression and an assignment that is executed if the expression evaluates to true. This representation is used for cells from both condition and event tables. As with the branches in switch nodes, the expressions associated with the cells represented in a table node must be disjoint. Also associated with each table node is a tag that indicates if the table is complete, meaning that exactly one of the logical expressions must be satisfied whenever the table node executes. This flag is initialized to true for condition tables. (One property of a condition table is that, in any state, exactly one logical expression in a cell is always true.) The flag can also be set to true if it is determined that a cell in the table will always execute. Table nodes have a single outgoing edge, indicating the next node to be executed.

Execution-Flow Graph Based Code Generation Method. OSCR generates optimized code in five phases, with all but the first and the last being optional. Phase 1 builds the execution-flow graph from the SCR specification. Phase 2 applies input slicing to this graph, removing switch and table nodes updating variables that cannot change for a given input. Phase 3 uses the graph to simplify table and switch nodes. Phase 4 uses output slicing to identify table and switch nodes that can never contribute to the output of the system and eliminates them. Finally, in Phase 5, source code is generated from the graph.

4. Building the Execution-Flow Graph

To build an execution-flow graph from a SCR specification, we first determine the set of mode classes in the specification. We then topologically sort the dependent variables in the specification, finding an order in which they can be correctly evaluated. A single header node corresponding to the initial modes of the system is created and added to a worklist. We then build subtrees of the execution-flow graph starting at each node in the worklist, and ending at header nodes reachable from that node. These subtrees contain switch and table nodes. When a new header node is created, it is added to the worklist. This process continues until the worklist is empty.

Finding Modes. The first step in building the execution-flow graph is to identify the mode classes in the specification. Normally, this is simple, given that the specification explicitly defines each

```
worklist = an empty list of header nodes.
start = a set of mappings between modeclasses and their
   initial modes.

find_header(start) — This creates the initial header node, and
   adds it to worklist.

while worklist is not empty:
   pop header from worklist
   the next node for header for all inputs =
      efg_build(topo_deps, modes from header, { })
done.


function find_header(state):
   if a header node is found for state:
      return it
   else:
      create a new header node for state
      add it to the worklist
      return it


function efg_build(deps, prestate, poststate):
   if deps is empty:
      return find_header(poststate)

   dep = the first item in deps
   new_deps = all other items in deps

   if dep in modeclasses:
      create a new switch node s

      for each predicate and assignment in table of dep:
         target = efg_build(new_deps, prestate,
            poststate ∪ { dep ↦ val })
         add a branch to s that performs assignment when
            predicate is true.
         add an edge for that branch from s to target

      target = efg_build(new_deps, prestate,
         poststate ∪ { dep ↦ prestate(dep) })
      the default edge for s is to target
      return s

   else:
      create a table node t from the table for dep
      the next node for t =
         efg_build(new_deps, prestate, poststate)
      return t
```

Figure 2. Pseudocode for an algorithm that builds an execution-flow graph. `mode classes` are the set of mode classes in the specification, while `topo_deps` is a list of the dependent variables in the specification, topologically sorted by poststate dependencies.

mode class and its modes. However, sometimes a mode class is not used with proper style. In this case, the mode class is reclassified as a term variable with an enumerated type. When mode classes are mentioned in the rest of this paper, the reference is to mode classes that have not been reclassified as terms.

Topological Sort of Dependent Variables. Next, all of the dependent variables are topologically sorted based on their *new-state dependencies*. Variable a depends on variable b in the new-state if b' is used in the computation of a'. The variables are topologically sorted to ensure that each variable is computed after all of the variables whose values it depends on have been computed. Such a topological sort is always possible for a well-formed SCR specification. Because the new-state dependencies are partially ordered, more than one topological ordering of the dependent variables is possible. We choose an ordering in which mode classes and variables they depend on are evaluated before other variables, as this maximizes the amount of information known during the simplification phase. Our method arbitrarily chooses a topological order satisfying this condition.

Initial Header Node. The next step determines the initial set of modes. These are easily determined from the specification. An initial header node corresponding to those modes is created and added to the header node worklist.

Worklist Algorithm. Finally, the rest of the graph is constructed. To accomplish this, a header node is extracted from the worklist, and a tree is constructed of execution-flow graph nodes representing all transitions out of that node. This process is repeated until all header nodes have been processed.

Recursively Building Trees. Trees of nodes are built recursively by `efg_build`, a function that takes as input a (possibly empty) list of dependent variables, the set of modes the system was in in the prestate, and a set of modes that the system will enter in the poststate. This function returns an execution-flow graph node, often creating a new node in the process.When called with the topologically-ordered list of dependent variables, the set of modes corresponding to a header node, and an *empty* set of poststate modes, the function returns an execution-flow graph node that can be executed next, for all inputs, when the system is at that header node. Three cases define `efg_build`, each returning one of the node types:

1. When the list of dependent variables is not empty, and the first dependent variable in the list is not a mode class, a table node is constructed from the SCR table defining the value of that variable. The single outgoing edge points to a node returned by `efg_build` and called with the rest of the list (that is, the list without the first node), and the same prestate and poststate modes.
2. When the first variable in the list is a mode class, a switch node is constructed. This node has a branch corresponding to each row of the mode class table, with the expression labeling each branch corresponding to the expression found in the mode transition table, and the assignment being the newly-entered mode. For each branch we create one outgoing edge to the result of `efg_build`, which is called with the rest of the list, the same prestate, and the poststate updated with a mapping of the mode class to the new mode. We also create a default edge that is executed when no branch is taken. This edge reaches the result of `efg_build`, called with the rest of the list, the same prestate, and the poststate updated by mapping the mode class to the mode it had in the prestate.
3. Calling `efg_build` with an empty list indicates the end of a transition. At this point, the header node must be found which represents the state in which the program waits for the next transition to begin. This is done by finding the header node corresponding to the set of poststate modes that `efg_build` is called with. This set contains one mode for each mode class, as all dependent variables must be processed before a header node can be created. If a header node corresponding to this set of poststate modes exists, it is returned. Otherwise, a new header node corresponding to these modes is added to the worklist, and then returned.

An empty worklist indicates that construction of the execution-flow graph is complete. While code can be directly generated from this graph, there is little advantage in doing so. Indeed, without further optimization, such code would almost certainly run more slowly, given that the graph is larger then the original specification by a factor quadratic in the number of combinations of modes in the specification. Representing modes in the graph makes information about modes explicit, which will allow optimization to begin.

The execution-flow graph in Figure 1 is the execution-flow graph that our method generates from the thermostat specification. It contains two header nodes, one for each of the two modes of `setting`. The node following each header node is a switch node representing the evaluation of `setting`. Three edges leave each switch node, two for the rows in the mode transition table, and one for the case in which the mode does not change. At this phase, the graph still contains impossible branches (such as a change from `enabled` to `enabled`). These branches will be eliminated during simplification. Along each of the branches leaving the switch nodes, table nodes computing `desired` and `heat` are executed before the header node corresponding to the new value of `setting` is reached. In the initial graph, new values are computed for every dependent variable along every path between header nodes. In the next three sections, we describe how many of these computations can be eliminated.

5. Input Slicing

Input slicing is a form of forward slicing [27] that eliminates parts of the program that do not depend on a variable of interest. To specialize the representations of the transform function based on the program inputs, our method applies input slicing. Input slicing allows us to improve performance by eliminating nodes that update variables that cannot change for given inputs. It also adds information to the execution-flow graph that can be exploited in later phases.

Particularly useful during the input slicing process are *update dependencies* [3]. The set of update dependencies of a dependent variable contains the smallest set of variables such that at least one of those variables changes every time the dependent variable changes. The update dependencies of a variable are always a subset of new-state dependencies. Event and mode transition tables may have sets of update dependencies that are smaller than their sets of new-state dependencies. Specifically, if e and f are expressions, then when an expression has the form (`not` e) `and` e' `and` f or the form e `!=` e' `and` f, the set of update dependencies of that expression only includes the variables e' depends on. Such expressions are created by translating SCR events into their logical equivalents; they may also be written explicitly. The set of update dependencies in a table is the union of the sets of update dependencies of the expressions in the cells in that table. These update dependencies can be used to determine the set of all variables that can change during a transition in which a particular monitored variable changes. If a variable is not in this set, then there is no need to compute the value of that variable.

The result of applying input slicing is that specialized update code is constructed for each input such that only variables that can change value are computed when the program receives that input. Update code which only computes variables that can change for a given input when at a given header node is an input slice. Computing an input slice for every combination of input and header nodes in the execution-flow graph is input slicing.

To compute an input slice for a given input and header node, we first compute the set of variables that are update-dependent, directly or indirectly, on that input. We then recursively copy the table and switch nodes reachable from that header node. Table and switch nodes are only copied if the variable they depend on is found in the set. If not, they are replaced by a copy of the tree reachable through the outgoing edge of the table node, or the default outgoing edge of a switch node. This process of copying with node elimination continues until a header node is reached. Header nodes are never copied, but instead new incoming edges to them are created.

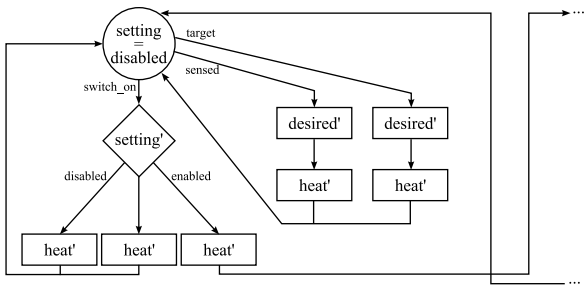

Figure 3. The execution-flow graph after applying input slicing to the graph of Figure 1. Edges leaving header nodes are now labeled with the inputs that cause them to be traversed. Half the graph is shown, the rest is similar.

Figure 3 shows the results of applying input slicing to the execution-flow graph created from the thermostat example. When `switch_on` changes, `desired` cannot change. Likewise, when `sensed` or `target` change, `setting` cannot change. Input slicing lets us eliminate one computation along every path between header nodes. Because it is rare for any input to cause an update of every dependent variable, input slicing generally reduces the amount of work needed. It does this by trading space, in the form of an increased number of update code variants, for speed. For large specifications, where the number of dependent variables that can change for any given input tends to be relatively small, input slicing can lead to substantial speedups. At the same time, we also encode into the execution-flow graph information about the input causing a transition, making it available to the next phase of the optimization process, simplification.

6. Simplification

Simplification uses information about modes and inputs encoded in the execution-flow graph, as well as other information supplied as assertions and assumptions, to reduce the complexity of expressions and thus the time needed to evaluate them. It can also eliminate entire cells and branches when it determines that there is no way they can occur, thus reducing the size of the execution-flow graph and the generated code. The simplification process works by keeping track of a number of substitutions that are known to be valid during the current transition. We apply substitutions and symbolically evaluate the expression until no further evaluation can be done. For example, if we have the expression $a' = b$, and the substitution $b \mapsto a'$, then applying this substitution will turn this expression into $a' = a'$, which is always true.

We choose a set of substitutions such that a simpler expression is always substituted for a more complex one. For our purposes, constants (including True, False, and mode class and enumeration values) are simpler than poststate variables, which are simpler than prestate variables, which are simpler than other expressions. Subject to this condition, we define simplicity as an arbitrary total ordering of expressions. To ensure that a substitution always produces an expression that is no more complex than the original, we disallow substitutions from one arbitrary expression to another, only allowing substitution from an expression to a variable or constant. Substitutions come from a number of sources. When a substitu-

tion has been created from a node or edge in the execution-flow graph, the substitution remains in effect until the next header node is reached. Substitutions can be derived from each of the following:

- **Assumptions and assertions.** The SCR toolset allows one to specify two kinds of properties: environmental assumptions, which are assumed to be true, and assertions, properties of the specification which have been proven to be true. The user specifies both assumptions and assertions. In addition, assertions may be automatically generated from the specification using the method described in [15, 17]. During simplification, assumptions and assertions are assumed to hold for all inputs and all state transitions. This allows us to introduce substitutions of the form $e \mapsto true$, where e is an invariant. These substitutions may then be further simplified to derive other substitutions.
- **Modes at header nodes.** Since each header node represents a combination of modes the system is in, during transitions all of the prestate modes are known. Based on this, substitutions of the form $mc \mapsto m$ are derived, where mc is a mode class, and m is the value of mc in the prestate . This allows us to simplify the mode comparisons in event and mode transition tables.
- **Inputs.** If we know, from input slicing, the input that caused a node to execute, we can derive two kinds of substitutions. By the One Input Assumption, a change in the value of one monitored variable implies that the values of all other monitored variables are unchanged. We also know that any variables not update-dependent on a given input cannot change. When a variable v cannot change during a transition, we can add a substitution of the form $v \mapsto v'$.
- **Switch nodes.** When taking a branch of the switch node, we can derive two more substitutions, $c \mapsto true$, and $mc' = m \mapsto true$, where c is the condition of the branch, mc' is the mode class the switch node computes, and m is the new mode. This simplifies to $mc' \mapsto m$ and thus allows the elimination of the mode expressions found in condition tables. Along the default edge, $mc \mapsto mc'$ is derived because the mode class cannot change. As we know the value of mc, we can derive a substitution for mc'.
- **Table nodes.** When simplification identifies a cell that must execute, either because its condition is true or all other cells are false and the table is complete, we derive two substitutions. The first is $c \mapsto true$, which states that the condition is true. This is only an interesting substitution if all other cells are false. The second is $v' = e \mapsto true$, where e is the expression contained in the cell and v' the variable computed by the table node.
- **Simplification of substitutions.** The simplifier can apply other substitutions and the symbolic evaluator to substitutions. This is a powerful rule for deriving new substitutions. For example, it allows us to determine that, if both $a \wedge b \mapsto true$ holds and $b \mapsto false$ holds, then $a \wedge false \mapsto true$, which simplifies to $a \mapsto true$.
- **Derivation rules.** We have formulated several rules for deriving substitutions from other substitutions. These rules include:
  - $a = b \mapsto true \longrightarrow not(a = b) \mapsto false$
  - $a = b \mapsto false \longrightarrow not(a = b) \mapsto true$
  - $a \mapsto true \longrightarrow not\ a \mapsto false$
  - $a = b \mapsto true \longrightarrow a \mapsto b$ or $a = b \mapsto true \longrightarrow b \mapsto a$

  Derivation or simplification may lead to a situation in which there exist two possible substitutions for the same expression. For example, $a \mapsto b$ and $a \mapsto c$. As a substitution can only exist if two values are equal, we can rewrite the substitutions as $a \mapsto c$, $b \mapsto c$, if c is the simpler of b and c.

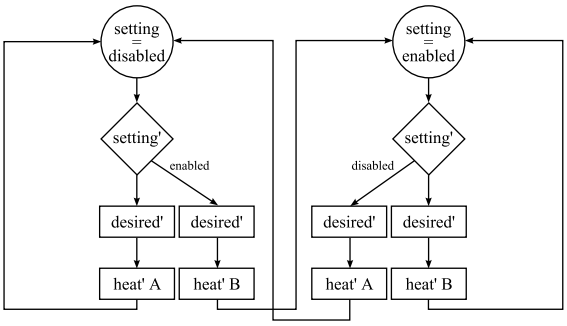

Figure 4. The execution-flow graph after simplifying the graph in Figure 1.

	True
heat =	False

Table 4. The A simplified variant table for `heat`, used when `setting' = disabled`.

	not `desired`	`desired`
heat =	False	True

Table 5. The B simplified variant table for `heat`, used when `setting' = enabled`.

The simplifier is applied to every condition and expression in every table and switch node in the graph. When applied to a condition, the simplifier is applied to each element of the outermost conjunction, from left to right. After simplifying expressions as much as possible, the conjunct is assumed true when simplifying later conjuncts. This eliminates redundancy from conditions. For example, $a \wedge (a \vee b)$ is simplified to just a. In simplifying a table node, cells containing an expression that evaluates to false are eliminated from the node entirely. If a cell's conjunction simplifies to true, that cell must execute, and hence (by the disjointness property satisfied by SCR tables) all other cells may not, and are eliminated from the table. Finding a true predicate also lets us to mark the table as complete, thus allowing us to derive more substitutions. Finding a false expression for a branch in a switch node allows us to remove that branch. An expression that evaluates to true allows us to eliminate all other branches, and the default edge, from that switch node. This allows us to remove large portions of the execution-flow graph.

Figure 4 shows the results of applying simplification to the execution-flow graph in Figure 1. (To reduce the size of the figure, we do not show a simplified input-sliced graph.) The branches of the switch nodes that cannot execute (because it is impossible to transition from a mode to the same mode) have been eliminated from the graph. We created two simplified variants of the table defining `heat`. Variant A, in Table 4, is used when `setting` is `disabled` in the poststate, while variant B, in Table 5, is used when `setting` is `enabled` in the poststate. Both of them are simpler than Table 3, the original table defining `heat`.

7. Output Slicing

Up to this point, the value of every variable that depends on a given input has been computed. However, if, for any series of inputs, the result of a computation can never influence the value of a controlled variable, this computation is unnecessary. Because every table in a well-formed SCR specification must influence the value of at least one controlled variable, this optimization is only possible after simplification which allows us to "shut off" portions of the system in certain modes. The process of eliminating computations that cannot affect a program's output is a form of backward slicing [27] called *output slicing*.

To perform output slicing, all prestate and poststate variables that are live at nodes of the execution-flow graph must be identified. A variable is *live* in a state if its value can eventually influence the value of a controlled variable. If a variable is not live in the poststate at a table node where its value is computed, then that node can be eliminated without affecting the externally visible behavior of the system.

To compute the set of live variables in each state at each node in the execution-flow graph, a backwards data-flow analysis is applied. We determine the poststate variables that must be live at each node, and, for each type of node in the graph, we specify a rule that determines the prestate and poststate variables that must be live at nodes having control-flow edges to that node. A workset algorithm is used to repeatedly apply these rules until a fixed point is reached, thus identifying the variables that are live at all nodes in the execution-flow graph. There are two cases where variables must be live in the poststate for slicing to be correct. First, each controlled variable must be live in the poststate at the table node that computes it. This is to ensure that output slicing will never eliminate a computation of a controlled variable, since this could change the behavior of the system. Likewise, each mode class must be live in the poststate at the switch node that computes it, thus ensuring that the execution flow of the system remains unchanged.

Below are three rules for propagating liveness through the execution-flow graph:

- If a variable is live in the poststate at a node that computes it, then all variables on which it depends (in the prestate or poststate) are live at all nodes with edges to that node. This ensures that calculations used by live calculations are also live. The dependencies used here are those recalculated after simplification is complete. After simplification, a node often has fewer dependencies than it had before simplification, making output slicing effective.
- Variables live at a node that does not compute a variable value are live at nodes with incoming edges to that node. This allows liveness to propagate through irrelevant nodes.
- If a variable is live in either the prestate or poststate at a header node, it is live in the poststate at nodes with edges to that header node. This is because header nodes represent boundaries between transitions. Variables that were live in the prestate in one transition are live in the poststate in the previous transition. Variables live in the poststate at a header node are variables with values that were not computed during that transition. As a result, they have the same value as they had in the transition's prestate, the poststate of the previous transition. A node with an edge to a header node will never have variables that are live in the prestate.

After applying these rules until a fixed-point is reached, every node is labeled with the variables live in its prestate and poststate. If a variable is not live in the poststate at a table node that computes its value, that node computes a value that is never used, and therefore is eliminated. Nodes eliminated in this way always compute the values of terms, and are often created from condition tables.

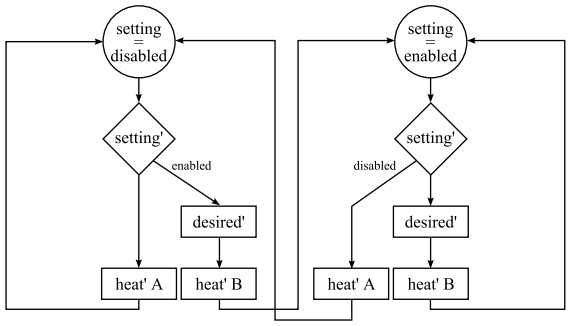

Figure 5. The execution-flow graph after output slicing Figure 4.

Figure 5 shows the result of applying output slicing to the simplified graph of Figure 4. We remove the computations of `desired` when `setting'` is `disabled`. This is because `desired` is only used by variant B of the table defining `heat`, and is always recomputed without reference to itself on paths where it is used.

Hybrid Slicing. Input slicing may reduce the effectiveness of output slicing. One case in which output slicing is particularly effective is when a variable is always recomputed (without using the prestate value of the variable) when entering modes in which that variable is used. Unfortunately, input slicing can eliminate such recomputations. In this case, the variable will need to be maintained in the old mode to ensure that the value of the variable is available when needed.

As a solution to this problem, we may perform *hybrid slicing*, which suppresses input slicing on paths through the execution-flow graph where a mode transition takes place. Specifically, when this *hybrid slicing* option is enabled, input slicing does not take place on paths from a branch in a switch node to the next header node. Input slicing does take place along the default edge leaving a switch node. By ensuring that all tables are recomputed during mode changes, hybrid slicing allows input and output slicing to coexist. When our assumption that inputs causing mode transitions are rare holds, hybrid slicing is often profitable.

8. Code Generation

After the execution-flow graph has been created and optimized, it can be used to generate code in a target implementation language. Code generation occurs in two steps. First, the execution-flow graph is used as the basis for creating an abstract syntax tree (AST) corresponding to the final code. Then the AST is used to generate target language code. As part of the first step, the system is converted to an event-driven form.

The generated system exposes an API based on method (or function) calls. For each monitored variable in the initial specification, a target-language method is created. When a monitored variable changes, we expect the corresponding *change method* to be called with the new value of the variable. We update the internal state of the system in response to these calls. We call *control methods* supplied by supporting code when the value of a controlled value changes. We also generate an `init` method that sets the state of the system to the initial values defined in the specification.

The AST created contains nodes for variable and method declarations, method invocations, if and switch statements, and variable assignment. This AST is sufficient to represent the generated program.

For each variable in the specification, the generated program contains two variables, one representing the variable in the prestate and the other the value in the poststate. During a transition, values are calculated for some or all poststate variables. After these updates, the prestate variables are updated to the new values of the poststate variables.

Another variable, named `megamode`, stores an integer representing the set of modes the system is in when it waits for input. This is equivalent to assigning an integer to each header node. By assigning these integers in a dense manner, we allow a switch statement on the megamode to be implemented as a table of jump addresses, efficiently dispatching to the code for the appropriate implementation of the transform function. This is unnecessary for specifications having no mode class.

When transforming the execution-flow graph into an event-driven program, one change method is created for each input. The outermost construct in the body of this method is a switch on `megamode`, the set of modes that the system is in. For each value of `megamode`, we find the corresponding header node in the execution-flow graph, and follow the edge from that node corresponding to the input. This yields another node. We recursively translate the tree of nodes rooted at that node into an AST, based on the kind of node encountered, as described below.

- Each **table node** is translated to an if-statement, with one branch per cell in the corresponding table. The condition of each branch is the expression contained in the cell, and the block corresponds to the assignment associated with the cell. For a complete table, the last branch can be converted to an else branch, as one cell must always run in a complete table. Otherwise, the else branch is omitted. The whole statement is followed by the translation of the single next node. If a table node computes a controlled variable, its translation includes code to call the appropriate control method when the variable changes value.
- Each **switch node** is also translated into an if-statement. The branches are conditioned by the expressions labeling the branches in the switch node and contain the appropriate assignment followed by a translation of the node reached through the outgoing edge corresponding to the branch. If a default outgoing edge exists, the translation of the node reached along this edge is the else branch. In this way, switch nodes generate multiple branches of execution.
- When reached, a **header node** marks the end of a transition, i.e., the end of a branch of execution. When a header node is encountered, we generate code to set `megamode` to the set of modes found in the next header, and to update prestate variables corresponding to changed poststate variables.

The AST is created as described above until all change methods have been created. Finally, the AST is used to generate target-language code in a straightforward manner.

9. Implementation and Experiments

To demonstrate the feasibility and effectiveness of our method, we have developed a tool, named OSCR, that automatically generates optimized code from SCR specifications. This tool also has several other capabilities in addition to code generation that help us experiment with the generated code and thus evaluate how useful, in practice, our method is.

9.1 Implementation

OSCR consists of over 5,400 lines of Python code. The bulk of this code is used to implement the execution-flow graph creation, optimization, and C/Java code generation phases of our method, as well as the additional capabilities described below. A small fraction of the code is used to implement other functionality, not described here, that stood to benefit from sharing the data structures and analyses we perform.

One additional capability of the OSCR tool is to generate random input for evaluating the performance of the generated code. Given a specification, OSCR can produce a stream of valid yet random inputs that can be supplied to the specification. The characteristics of a random input stream may actually lead to results inferior to those produced from more structured input streams. Random input often contains large jumps in variable values, and frequent changes in the value of variables that would in practice rarely change. In the running thermostat example, the `switch_on` and `target` variables rarely change, while the `sensed` temperature would change far more often. Random input creates a similar number of events for all three inputs. This can cause overly-frequent mode changes, which may cause our experiments to underreport the benefits of our method.

The tool can also generate framework programs for testing and profiling the generated code. It takes as input a specification and a stream of inputs, and generates a framework that supplies that input to the specification a given number of times. Output can either be printed for testing purposes, or ignored to allow for profiling. The input is actually converted into a series of functions calls, as parsing the input at runtime would entail overhead that would obscure the running time of the generated code.

The amount of time OSCR takes to generate optimized code from a specification is largely determined by the size of the specification. It requires only a few seconds to process a small specification, while our largest specification, containing 1,114 tables, takes less than sixteen hours. Most of that time is spent in the simplification phase. The other phases all complete in seconds, even for our largest specification.

OSCR also implements a number of potential optimizations that were considered and rejected. One such optimization uses flags to indicate which variables actually need updates. Despite its use in other programs (such as the simulator described in [12]), we found this optimization to be largely ineffective due to the high cost of updating and checking the flags.

9.2 Experiments

To evaluate the effectiveness of our methods, we have used OSCR to automatically generate code for a number of specifications, ranging from small synthetic examples (such as the running thermostat example) to large flight control systems developed by industry. Table 6 describes the specifications with which we have experimented. The two variants of the safety injection system are present to show how the optimization process can take pre-existing invariants into account.

Table 7 shows the running times of C code generated from SCR specifications using various combinations of the optimizations described above. It also compares the running times to a baseline variant created without use of an execution-flow graph, generated using a reimplementation of the method in [21]. Empty cells represent optimizations that could not be applied, due to the lack of mode classes in the wcp and acs specifications. In all cases, a maximal combination of the effective optimizations, either with or without hybrid slicing, produced the best results. This is to be expected, as the random input data produces frequent mode changes, which hurts the performance of the hybrid-sliced variant more than the other variants. While input slicing produced the largest speedups, output slicing and simplification also produced substantial performance improvements.

name	MVs	DVs	modes	description
ts	3	3	2	The thermostat running example.
sis	3	3	3	The safety injection system is found in [13].
sisinv	3	3	3	The same safety injection system, with automatically-generated invariants.
cc	10	12	4	A cruise control system example.
scram	26	4	3	Shutdown control logic for a nuclear power plant.
wvp	74	159	1	Weapons control panel.
acs	230	1114	1	Aircraft control software.

Table 6. Descriptions of benchmark specifications. MVs is number of monitored variables, DVs the number of dependent variables.

variant	ts	sis	sisinv	cc	scram	wcp	acs
baseline	3.87 (1.00)	5.21 (1.00)	4.96 (1.00)	15.52 (1.00)	11.42 (1.00)	5.64 (1.00)	44.81 (1.00)
simplify	3.87 (1.00)	3.71 (1.40)	3.63 (1.37)	10.05 (1.54)	4.90 (2.33)	5.42 (1.04)	43.49 (1.03)
simplify/os	3.87 (1.00)	3.56 (1.46)	3.60 (1.38)	9.11 (1.70)	4.60 (2.48)	5.37 (1.05)	40.05 (1.12)
is/simplify	3.22 (1.20)	2.78 (1.87)	2.45 (2.02)	5.66 (2.74)	1.73 (6.60)	1.65 (3.42)	0.69 (64.94)
is/simplify/os	3.22 (1.20)	2.82 (1.85)	2.44 (2.03)	5.45 (2.85)	1.73 (6.60)	1.60 (3.52)	0.62 (72.27)
simplify/hybrid	2.93 (1.32)	2.73 (1.91)	2.56 (1.94)	5.53 (2.81)	1.78 (6.42)		

Table 7. Running times of optimization variants, as cpu time in seconds. Numbers in parenthesis are relative speedups. Empty cells represent inapplicable optimizations. “is” stands for input slicing, and “os” for output slicing.

variant	ts	sis	sisinv	cc	scram	wcp	acs
baseline	2.6 (1.00)	1.5 (1.00)	1.5 (1.00)	3.9 (1.00)	4.4 (1.00)	29.1 (1.00)	311.5 (1.00)
simplify	1.4 (0.54)	2.0 (1.27)	1.8 (1.23)	9.9 (2.52)	6.4 (1.47)	28.6 (0.98)	307.4 (0.99)
simplify/os	1.4 (0.54)	1.9 (1.24)	1.8 (1.22)	8.1 (2.06)	6.2 (1.42)	28.4 (0.97)	278.5 (0.89)
is/simplify	3.1 (1.21)	3.8 (2.46)	3.7 (2.50)	19.8 (5.02)	13.8 (3.16)	158.0 (5.42)	1004.7 (3.23)
is/simplify/os	3.1 (1.21)	3.8 (2.47)	3.7 (2.49)	17.6 (4.47)	13.8 (3.16)	157.8 (5.41)	926.6 (2.97)
simplify/hybrid	2.9 (1.12)	3.5 (2.31)	3.5 (2.37)	15.6 (3.94)	12.1 (2.77)		

Table 8. Object sizes, in kilobytes, with relative sizes in parentheses. Empty cells represent inapplicable optimizations.

In no case did optimization decrease performance, and, in many cases, it more than doubled performance. In general, the amount of speedup increases with the complexity of the specification. We believe that this is because the larger a specification is, the more likely it is that computations will exist that do not need to be executed for any given input and set of modes. This is borne out by the largest performance increase (over 72 times) gained in applying our optimizations to the largest specification (acs, with 1,114 tables).

Table 8 shows the object code size for each variant. The reported sizes are determined by taking the size of the stripped object file compiled from the generated code. The reported sizes include the size of the initialization code, which runs only once, and therefore need not be considered when understanding long-term cache behavior. An ineffective optimization (always backward slicing) leads to a variant with the same size as another variant.

Interestingly, we never encounter a specification that has a worst-case size increase. The worst case size increase for representing the modes in the execution-flow graph is equal to the square of the number of combinations of modes in the system. We never encounter this, with the largest size increase due to simplification equal to a factor of 2.52 for the cc specification, which has 4 modes. Likewise, input slicing could potentially expand the specification by a factor of the number of monitored variables in the specification. Instead, the largest increase is by a factor of 5.42, on a specification with 74 variables. The simplification allowed by representing inputs and modes in the execution-flow graph makes up for much of the potential cost in object size.

The specifications sis and sisinv differ only in the presence or absence of additional invariants, generated using the method described in [15, 17]. Table 7 shows that when input slicing is used, the presence of these invariants leads to a noticeable performance improvement.

10. Related Work, Future Work, and Conclusion

In this paper, we have described execution-flow graphs, a representation of state-machine based specifications that supports code generation and optimization. We have also described three optimizations—input slicing, simplification, and output slicing—that exploit properties of these specifications. A tool, OSCR, was introduced that can automatically apply these techniques. Our experimental results demonstrate the effectiveness of our techniques as applied to SCR specifications. Adding the tool OSCR to the SCR toolset will allow SCR users to use our method to generate optimized code from SCR specifications.

This work was motivated by research which used the APTS program transformation system to perform unoptimized code generation from SCR specifications [21]. OSCR improves on this work in three ways: It is fully automatic, not requiring a manual translation step; it is much faster, taking seconds to generate unoptimized code from a specification with 1,114 tables (rather than 12 hours to generate code for a specification with 20 tables); and it can generate optimized code.

The work most similar to ours is the code generation work performed for LUSTRE [9]. The LUSTRE code generation method involves building a state machine representing the boolean variables

in the specification, and then producing simplified implementations of the transform functions for each state. It was abandoned by the Esterel toolchain (used by the LUSTRE compiler) due to a state explosion problem (mentioned in [6]). Our synthesis method exploits SCR mode classes, which are few in number and important, allowing us to avoid state explosion. While some work has been done on slicing LUSTRE [8] and Esterel [19], this work is for program understanding rather than optimization.

Code generation has been developed for other specification languages, such as RSML [28], Input/Output Automata [26], and Charon [1]. Reference [28] mentions that the code generator for RSML performs input slicing, but does not go into detail. The code generators for the other two languages do not seem to perform any of the optimizations described in this paper.

By themselves, none of the optimizations that OSCR performs are novel. Generating specialized update code for particular transitions is a form of incrementalization [24]. Program slicing is a well-known technique [27], and backward slicing has been applied by others to program optimization [25]. Input slicing for optimization is mentioned in a paragraph in [28] but its value is diminished by the requirement for discrete input types. Simplification based on known variable values is described in [9]; our technique for simplification is a form of constraint-based partial evaluation [20]. This paper shows that these techniques complement each other in a manner suitable for generation of optimized code from SCR specifications. The One Input Assumption makes input slicing possible by reducing the possible inputs to a manageable size, while the limited number of mode classes in high-quality SCR specifications improves the success of backward slicing and simplification.

Our current method improves the performance of a program while maintaining the One Input Assumption—i.e., allowing only one monitored event to change from one state to the next. One possible avenue of future work is to study optimizations that discard this assumption, allowing us to process several inputs in a single transition. A second avenue of future work is formal proof of the correctness of the methods described herein.

Acknowledgments. We thank Myla Archer for her comments on an earlier version of this paper. We also are grateful to Myla, Michael Colón, Ralph Jeffords and everyone else in NRL code 5546 for very useful discussions while this research was in progress.

References

[1] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. *SIGPLAN Not.*, 38(7):171–182, 2003.

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

[3] M. Archer. Basing a modeling environment on a general purpose theorem prover. In *Proc. Monterey Workshop on Software Engineering Tools: Compatibility and Integration*, Baden, Austria, October 4-6 2004.

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

[6] S. A. Edwards. An Esterel compiler for large control-dominated systems. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.

[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] F. Gaucher. Slicing LUSTRE programs. Technical report, VERIMAG, Grenoble, February 2003.

[9] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[10] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering*, 20(1):19–35, Jan. 2005.

[11] 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.

[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 Trans. on Software Eng. 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] 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.

[16] R. Jeffords and E. Leonard. Using invariants to optimize formal specifications before code synthesis. In *Proc. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004)*, June 2004.

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

[18] 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.

[19] A. R. Kulkarni and S. Ramesh. Static slicing of reactive programs. In *SCAM*, pages 98–107, 2003.

[20] L. Lafave and J. P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In *LOPSTR '97: Proc. of the 7th International Workshop on Logic Programming Synthesis and Transformation*, pages 168–188, London, UK, 1998. Springer-Verlag.

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

[22] 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.

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

[24] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL '93: Proc. of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 502–510, New York, NY, USA, 1993. ACM Press.

[25] T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.

[26] J. A. Tauber, N. A. Lynch, and M. J. Tsai. Compiling IOA without global synchronization. In *NCA '04: Proc. of the Network Computing and Applications, Third IEEE Int'l Symp. on (NCA'04)*, pages 121–130, Washington, DC, USA, 2004. IEEE Computer Society.

[27] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[28] M. W. Whalen. High-integrity code generation for state-based formalisms. In *ICSE '00: Proc. of the 22nd Int'l Conf. on Software Eng.*, pages 725–727, New York, NY, USA, 2000. ACM Press.