

Using Model Checking to Generate Tests from Requirements Specifications*

Angelo Gargantini¹ and Constance Heitmeyer²

¹ Politecnico di Milano, Milano, Italy

Angelo.Gargantini@elet.polimi.it

² Code 5546, Naval Research Laboratory, Washington DC 20375

heimeyer@itd.nrl.navy.mil

Abstract. Recently, many formal methods, such as the SCR (Software Cost Reduction) requirements method, have been proposed for improving the quality of software specifications. Although improved specifications are valuable, the ultimate objective of software development is to produce software that satisfies its requirements. To evaluate the correctness of a software implementation, one can apply black-box testing to determine whether the implementation, given a sequence of system inputs, produces the correct system outputs. This paper describes a specification-based method for constructing a suite of *test sequences*, where a test sequence is a sequence of inputs and outputs for testing a software implementation. The test sequences are derived from a tabular SCR requirements specification containing diverse data types, i.e., integer, boolean, and enumerated types. From the functions defined in the SCR specification, the method forms a collection of predicates called *branches*, which “cover” all possible software behaviors described by the specification. Based on these predicates, the method then derives a suite of test sequences by using a model checker’s ability to construct counterexamples. The paper presents the results of applying our method to four specifications, including a sizable component of a contractor specification of a real system.

1 Introduction

During the last decade, numerous formal methods have been proposed to improve software quality and to decrease the cost of software development. One of these methods, the SCR (Software Cost Reduction) method, is based on a user-friendly tabular notation and offers several automated techniques for detecting errors in software requirements specifications, including an automated *consistency checker* to detect missing cases and other application-independent errors [14]; a *simulator* to symbolically execute the specification to ensure that it captures the users’ intent [13]; and a *model checker* to detect violations of critical application properties [3, 12]. Recently, groups at NASA and Rockwell Aviation as well as our group at NRL have used the SCR techniques to detect serious errors in requirements specifications of real-world systems [7, 21, 12]. By exposing defects in the requirements specification, such techniques help the user improve the specification’s quality. This improved specification provides a solid foundation for the later phases of the software development process.

While high-quality requirements specifications are clearly valuable, the ultimate objective of the software development process is to produce high-quality *software*, i.e., software that satisfies its requirements. To weed out software errors

* This research is funded by the Office of Naval Research and SPAWAR.

and to help convince customers that the software performance is acceptable, the software needs to be tested. An enormous problem, however, is that software testing, especially of safety-critical systems, is extremely costly and time-consuming. It has been estimated that current testing methods consume between 40% and 70% of the software development effort [2].

One benefit of a formal method is that the high-quality specification it produces can play a valuable role in software testing. For example, the specification may be used to automatically construct a suite of test sequences. These test sequences can then be used to automatically check the implementation software for errors. By eliminating much of the human effort needed to build and to apply the test sequences, such an approach should reduce both the enormous cost and the significant time and human effort associated with current testing methods.

This paper describes an original method for generating test sequences from an operational SCR requirements specification containing mixed variable types, i.e., integers, booleans, and enumerated types. In our approach, each *test sequence* is a sequence of system inputs and their associated outputs [19]. The requirements specification is used both to generate a valid sequence of inputs and as an *oracle* [17] that determines the set of outputs associated with each input. To obtain a valid sequence of inputs, the input sequence is constrained to satisfy the input model (i.e., assumptions about the inputs) that is part of the requirements specification. Our method for generating test sequences “covers” the set of all possible input sequences by organizing them into equivalence classes and generating one or more test sequences for each equivalence class.

Section 2 reviews the SCR method, and Section 3 describes the objectives of an effective suite of test sequences. After showing how test sequences can be derived from system properties, Section 4 presents our original method for generating test sequences from operational requirements specifications and the branch coverage criterion that the method applies. Section 5 describes a tool we developed that uses either of two model checkers to automatically generate test sequences; it also presents the results of applying the tool to four specifications. Section 6 reviews related work, and Section 7 presents a summary and plans for future work.

2 Background: The SCR Requirements Method

The SCR method was formulated in 1978 to specify the requirements of the Operational Flight Program (OFP) of the U.S. Navy’s A-7 aircraft [15]. Since then, many industrial organizations, including Bell Laboratories, Grumman, Ontario Hydro, and Lockheed have used SCR to specify the requirements of practical systems. The largest application to date occurred in 1994 when Lockheed engineers used SCR tables to document the complete requirements of Lockheed’s C-130J OFP [9], a program containing more than 250K lines of Ada. Each of these applications of SCR had, at most, weak tool support. To provide powerful, robust tools customized for the SCR method, we have developed the SCR toolset, which includes the consistency checker, simulator, and model checker mentioned above. To provide formal underpinnings for the tools, we have formulated a formal model which defines the semantics of SCR requirements specifications [14].

An SCR requirements specification describes both the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [14]. The SCR model represents the environmental quantities that

the system monitors and controls as *monitored* and *controlled variables*. The environment nondeterministically produces a sequence of input events, where an *input event* signals a change in some monitored quantity. The system, represented in the model as a state machine, begins execution in some initial state and then responds to each input event in turn by changing state and by possibly producing one or more output events, where an *output event* is a change in a controlled quantity. An assumption of the model is that at each state transition, exactly one monitored variable changes value. To concisely capture the system behavior, SCR specifications may include two types of auxiliary variables, *mode classes*, whose values are *modes*, and *terms*. Mode classes and terms often capture historical information.

In the SCR model, a system is represented as a 4-tuple, (S, S_0, E^m, T) , where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is the transform describing the allowed state transitions [14]. Usually, the transform T is deterministic, i.e., a function that maps an input event and the current state to a new state. To construct T , we compose smaller functions, each derived from the two kinds of tables in SCR requirements specifications, event tables and condition tables. These tables describe the values of each *dependent variable*, that is, each controlled variable, mode class, or term. The SCR model requires the entries in each table to satisfy certain properties. These properties guarantee that all of the tables describe total functions.

In SCR, a state s is a function that maps each variable in the specification to a type-correct value, a *condition* is a predicate defined on a system state, and an *event* is a predicate defined on a pair of system states implying that the value of at least one state variable has changed. When a variable changes value, we say that an event “occurs”. The expression “@T(c) WHEN d” represents a *conditioned event*, which is defined by

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

where the unprimed conditions c and d are evaluated in the *current* state and the primed condition c' is evaluated in the *next* state.

3 Attributes of an *Effective Suite* of Test Sequences

A practical method should be supported by “pushbutton” techniques, techniques that can be invoked with the mere push of a button. One example of a pushbutton technique is automated consistency checking [14]. Our goal is to also make software testing a pushbutton technique, i.e., as automatic as possible. Our approach to software testing focuses on *conformance* testing, black-box testing that determines whether an implementation exhibits the behavior described by its specification. This approach divides testing into two phases. During the first phase, an operational requirements specification is used to automatically construct a suite of test sequences. During the second phase, a test *driver* feeds inputs from the test sequences to the software implementation, and then a *comparator* compares the outputs produced by the implementation with the outputs predicted by each test sequence, reporting all discrepancies between the two sets of outputs. Clearly, discrepancies between the two sets of outputs expose cases in which the software implementation violates the requirements specification.

The challenge of software testing methods is to produce an *effective* suite of test sequences. Like Fujiwara et al. [10], we believe that an effective suite of test sequences satisfies two (conflicting) objectives:

- The number of test sequences in the suite should be small. Similarly, the number of test data (i.e., the length of the input sequence) in each test sequence should also be small.
- The test suite should “cover” all errors that any implementation may contain. That is, it should evaluate as many of the different possible behaviors of the software as possible.

In our approach, each test sequence is a complete scenario, which starts in a legal initial state and which contains, at each state transition, a valid system input coupled with a set of valid system outputs. Both the assumptions that constrain the system inputs and how the system outputs are computed from the system inputs are described by an SCR requirements specification.

4 Generating Test Sequences with a Model Checker

Normally, a model checker is used to analyze a finite-state representation of a system for property violations. If the model checker analyzes all reachable states and detects no violations, then the property holds. If, in contrast, the model checker finds a reachable state that violates the property, it returns a “counterexample,” a sequence of reachable states beginning in a valid initial state and ending with the property violation. We use model checking not for verification nor to detect specification errors but, like some others [1, 5, 8], to construct test sequences. Like these others, we base our method on two ideas. First, the model checker is used as an oracle to compute the expected outputs. Second, the model checker’s ability to generate counterexamples is used to construct the test sequences. To force the model checker to construct the desired test sequences, we use a set of properties called *trap properties*.

Section 4.1 describes how trap properties are derived from system properties provided by designers or customers. Then, Section 4.2 describes an original extension of this method which derives trap properties systematically and automatically from an operational SCR requirements specification. Deriving trap properties in this manner ensures that the test sequences “cover” all possible behaviors described by the specification. To demonstrate that our method can be used with different model checkers, we use two different model checkers to construct the test sequences. The example presented in Section 4.1 uses the symbolic model checker SMV [20], whereas the example presented in Section 4.2 uses the explicit state model checker Spin [16]. To translate an SCR specification into either the language of Spin or the language of SMV, we use the translation method described in [3]. A Spin specification and an SMV specification obtained from an SCR specification using this translation method are semantically equivalent. Section 4.3 describes our coverage criterion, a form of *branch* coverage.

4.1 Generating Test Sequences from Properties

To introduce our method, we illustrate how the model checker SMV may be used to obtain a test sequence from a system property and an SCR requirements specification. We consider a system called the Safety Injection System (SIS), a

simplified version of a control system for safety injection in a nuclear plant [6], which monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may override safety injection by turning a “Block” switch to “On” and may reset the system after blockage by setting a “Reset” switch to “On”.

To specify the SIS requirements in SCR, we represent the SIS inputs with the monitored variables `WaterPres`, `Block`, and `Reset` and the single SIS output with a controlled variable `SafetyInjection`. The specification also includes two auxiliary variables, a mode class `Pressure`, an abstract version of `WaterPres`, and a term `Overridden` which indicates when safety injection has been overridden. An important component of the SIS specification is the input model for `WaterPres`, which constrains `WaterPres` to change by no more than 3 psi¹ from one state to the next. The input model that describes the possible changes to `WaterPres` is defined by $\{(w, w') : |w' - w| \leq 3, 0 \leq w, w' \leq 30\}$, where w represents the value of `WaterPres` in one state and w' represents its value in the next state. In this example, a constant `Low=10` defines the threshold that determines when `WaterPres` is in an unsafe region.

Suppose that we have verified that the operational SCR specification of SIS satisfies a safety property called P , which is defined by

`@T(WaterPres < Low) WHEN Block = On ^ Reset = Off => SafetyInjection' = Off.`

Property P states that if `WaterPres` drops below the constant `Low` when `Block` is `On` and `Reset` is `Off`, then `SafetyInjection` must be `Off`.

We can use the property P and the operational SCR specification of the SIS to construct a test sequence as follows. First, the operational specification is translated into the SMV language in the manner described in [3]. If our goal was to verify P , we would next translate P into CTL, the temporal logic of SMV. Because our goal is not to verify P but to construct a test sequence from P , instead we translate the *negation* of P 's premise into CTL, i.e.,

`AG!(EX(WaterPres<Low) & ! WaterPres<Low & Block = On & Reset = Off),`

where `AG!` represents ‘never’, `EX` represents ‘next’, and `!` represents negation. Because the negation of P 's premise is false in the SCR specification, running SMV detects a violation. To demonstrate the violation, SMV produces a counterexample, i.e., a trace of input events which starts in a valid initial state and ends when a violation of the CTL property is detected. This trace provides the basis for the desired test sequence. The CTL property is an example of a trap property.

Table 1 illustrates the test sequence that can be constructed from the counterexample produced when SMV detects a violation of the above trap property in the SIS specification. In the table, the initial values of `WaterPres`, `Block`, `Reset`, `SafetyInjection`, and `Pressure` are shown in step 0, which represents the initial state. (Due to lack of space, Table 1 omits the term `Overridden`.) To clarify which variable values change from one state to the next, Table 1 only shows the variable values which change at each step and omits the values that remain the same. Note that the changes in `WaterPres` from one state to the next never exceed 3 psi and thus satisfy the constraints of the input model.

¹ The abbreviation “psi” represents “pounds per square inch.”

Step No.	Monitored Var. Value	Controlled Var. Value	Mode Class Value
0	WaterPres=2 Block=Off Reset=On	SafetyInjection=On	Pressure=TooLow
1	Reset=Off	SafetyInjection=Off	Pressure=Permitted
2	WaterPres=5		
3	WaterPres=8		
4	WaterPres=10		
5	Block=On		
6	WaterPres=8		

Table 1. Test Sequence Constructed from SMV Counterexample.

The six inputs that lead to the violation of the trap property form the input sequence for the test sequence. Table 1 shows that the only change to the SIS output produced by this input sequence is the change at step 4 in the value of `SafetyInjection`.

The test sequence of length six shown in Table 1 may be represented more concisely as

$$\langle (r, \text{off}; -), (w, 5; -), (w, 8; -), (w, 10; s, \text{off}), (b, \text{on}; -), (w, 8; -) \rangle, \quad (1)$$

where r , w , and b represent the input variables `Reset`, `WaterPres`, and `Block`; s represents the single output variable `SafetyInjection`; and $-$ indicates that no output variable changes.² Clearly, checking the software behavior with this test sequence will test whether the software satisfies property P . In addition to changes in output values, a test sequence may also include changes in the values of one or more auxiliary variables. For example, the test sequence in (1) could be extended to include changes in the mode class `Pressure`, which changes (see Table 1) to `Permitted` at step 4 and to `TooLow` at step 6.

Although this method can test many critical aspects of the system behavior, it has several weaknesses. First, the method assumes that the customers (or the designers) have formulated a set of system properties. Unfortunately, formulating such properties is not a normal step in requirements specification, and hence such properties may not be available. Second, and more important, is the incompleteness of the test sequences. Even if a large set of properties are available for generating test sets, questions remain about how completely the test sequences cover all possible system behaviors. Finally, the method assumes the correctness of both the operational specification and the properties. Our experience with SCR specifications and the SCR tools convinces us that achieving a high-quality SCR specification is feasible. In contrast, verifying that the specifications satisfy a given set of properties is more problematic. Although there has been recent progress in using model checkers and theorem provers to verify properties, formal verification still suffers from both theoretical problems (e.g., problems of decidability) and practical problems (time and space necessary for the proofs).

² In this example, the initial state is unique and thus may be omitted from the test sequence.

4.2 Generating Test Sequences from an Operational Specification

This section describes an original method for constructing test sequences which does not depend on a set of system properties. This method automatically translates an operational requirements specification in the SCR notation to the language of a model checker and automatically and systematically generates test sequences. This section first describes how test sequences can be generated from an event table and next how they can be generated from a condition table.

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

Table 2. Event Table Defining the Mode Class Pressure.

Generating Test Sequences from an Event Table. To illustrate the method, we consider the event table in the SIS specification (see Table 2) which defines the value of the mode class `Pressure`. The mode class has three modes: `TooLow`, `Permitted`, and `High`. At any given time, the system must be in one and only one of these modes. A drop in water pressure below the constant `Low` causes the system to enter mode `TooLow`; an increase in pressure above a larger constant `Permit=20` causes the system to enter mode `High`. Figure 1 shows the function that can be derived from Table 2 using the definition in [14]. The *else* clause in Figure 1 indicates that events not explicitly named in the table do not change the value of the variable being defined. To make the set of test sequences complete, we need to construct test sequences not only for the cases described explicitly in the table but for these “no-change” cases as well.

```

if
  □ Pressure = TooLow
    ∧ @T(WaterPres  $\geq$  Low)    -> Pressure' = Permitted
  □ Pressure = Permitted
    ∧ @T(WaterPres  $\geq$  Permit) -> Pressure' = High
  □ Pressure = Permitted
    ∧ @T(WaterPres  $<$  Low)     -> Pressure' = TooLow
  □ Pressure = High
    ∧ @T(WaterPres  $<$  Permit)  -> Pressure' = Permitted
  □ (else)                    -> Pressure' = Pressure
fi

```

Fig. 1. Function Defining Pressure With a Single *else* Clause.

To produce an interesting suite of test sequences for the no-change cases, we replace the definition of the mode class `Pressure` in Figure 1 with the equivalent definition in Figure 2, which associates an *else* clause with each possible value of `Pressure`. In Figure 2, the no-changes cases are labeled *C2*, *C5*, and *C7*. In each case, the value of `Pressure` does not change. Further, in each case, either the monitored variable `WaterPres` changes (in a way that satisfies the constraints

```

if
  □ Pressure = TooLow
    if
      □ @T(WaterPres ≥ Low) -> Pressure' = Permitted    C1
      □ (else) -> Pressure' = Pressure                    C2
    fi
  □ Pressure = Permitted
    if
      □ @T(WaterPres ≥ Permit) -> Pressure' = High      C3
      □ @T(WaterPres < Low) -> Pressure' = TooLow       C4
      □ (else) -> Pressure' = Pressure                  C5
    fi
  □ Pressure = High
    if
      □ @T(WaterPres < Permit) -> Pressure' = Permitted C6
      □ (else) -> Pressure' = Pressure                  C7
    fi
fi

```

Fig. 2. Function Defining Pressure With One *else* Clause per Mode.

of that case) or `WaterPres` does not change but another monitored variable (in this example, either `Block` or `Reset`) changes. In our approach, each of the C_i labels a “branch,” i.e., an equivalence class of state transitions. Together, these branches cover the interesting state transitions, i.e., those that change the value of the variable that the table defines and those that do not. Our approach is to construct one or more test sequences for each branch. For example, for the branch labeled $C1$, one or more test sequences will be constructed that satisfy both the hypothesis and the conclusion of the property

$$\text{Pressure} = \text{TooLow} \wedge @T(\text{WaterPres} \geq \text{Low}) \rightarrow \text{Pressure}' = \text{Permitted}.$$

To translate from the tabular format into Promela, the language of Spin, we apply the translation method described in [3]. Figure 3 shows the translation into Promela of the two branches labeled $C1$ and $C2$ in Figure 2. Because Promela does not allow expressions containing both “current” and “primed” values of variables, two Promela variables are assigned to each SCR variable. In Figure 3, each variable with a suffix of “P” represents a primed variable. We also translate each event in an event table to a Promela `if` statement. Thus, the event in the first row of Table 2 (labeled $C1$ in Figure 2) is translated to the `if` statement in lines 4–5 of the Promela code in Figure 3. In addition, we translate each *else* clause to a corresponding `else` statement in Promela. Thus, the *else* clause in the branch labeled $C2$ in Figure 2 is translated to a corresponding `else` statement in the sixth line of the Promela code in Figure 3.

To label the different cases³ in the Promela code, we introduce an auxiliary integer-valued variable `Casevar`, where *var* names the variable defined by the table. To indicate the case corresponding to C_i , we assign `Casevar` the value i .⁴ In this manner, we ensure that every branch assigns `Casevar` a dif-

³ Henceforth, this paper uses the terms “branch” and “case” interchangeably.

⁴ The introduction of an auxiliary variable is unnecessary and used here solely for clarity and generality. Our method works just as well without the use of new variables. For example, we may use Promela’s `goto` statement or similar alternatives.

```

if
  :: (Pressure == TooLow) ->
    if
      :: (WaterPresP >= Low) && ! WaterPres >= Low
        -> PressureP = Permitted; CasePressure = 1;
      :: else
        CasePressure = 2;
    fi
  ...
fi

```

Fig. 3. Promela Code for Cases *C1* and *C2*.

ferent value. Thus, in Figure 3, which shows two branches, the first branch is labeled `CasePressure = 1` and the second `CasePressure = 2`.

Next, we use Spin to construct one or more test sequences for each branch. To that end, we define a trap property which violates the predicate in the case statement. To construct a test sequence that corresponds to `CasePressure = 1` in Figure 3, we construct the negation of `CasePressure = 1` and insert it into a Promela `assert` statement:

```
assert (CasePressure != 1).
```

Then, any trace violating this trap property is a trace which satisfies the case (i.e., branch) with which `CasePressure = 1` is associated.

When Spin analyzes the SIS specification for the above trap property, it produces, as expected, a counterexample. From this counterexample, our method derives the test sequence of length 20 shown in Table 3. As required, the last two states of the test sequence satisfy the predicate labeled *C1* in Figure 2. That is, the sequence concludes with two states (s, s') such that, in state s , `WaterPres` $\not\geq$ `Low` and `Pressure` is `TooLow` (implied by `WaterPres = 9` at step 19) and, in state s' , `WaterPres` \geq `Low` and `Pressure` is `Permitted` (implied by `WaterPres` equals 10 at step 20). Moreover, as required by the input model, the value of `WaterPres` from one state to the next never exceeds three psi. One problem with this test sequence is its excessive length. Section 5 discusses this problem and shows how a much shorter test sequence may be built using SMV.

Step No.	Monitored Var. Value	Controlled Var. Value	Step No.	Monitored Var. Value	Controlled Var. Value
1	Block On		11	WaterPres 4	
2	Reset Off		12	Block On	SafetyInjection Off
3	Block Off		13	Block Off	
4	Block On	SafetyInjection Off	14	Reset On	SafetyInjection On
5	Block Off		15	Block On	
6	WaterPres 3		16	Reset Off	
7	Block On		17	WaterPres 5	
8	Reset On	SafetyInjection On	18	WaterPres 6	
9	Block Off		19	WaterPres 9	
10	Reset Off		20	WaterPres 10	SafetyInjection Off

Table 3. Test Sequence Derived from Spin Counterexample for Case *C1* of Fig. 2.

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

Table 4. Condition Table defining SafetyInjection.

Generating Test Sequences from a Condition Table. Test sets are generated from condition tables in a similar manner. For example, consider the condition table in Table 4, which defines the controlled variable `SafetyInjection`. Table 4 states, “If `Pressure` is `TooLow` and `Overridden` is *true*, or if `Pressure` is `Permitted` or `High`, then `Safety Injection` is `Off`; if `Pressure` is `TooLow` and `Overridden` is *false*, then `Safety Injection` is `On`.” The entry “False” in the rightmost column means that `Safety Injection` is never `On` when `Pressure` is `Permitted` or `High`. This table generates the four cases, *C1*, *C2*, *C3*, and *C4*, shown in Figure 4. Because condition tables explicitly define total functions (they never contain implicit no-change cases), there is no need to generate additional branches containing *else* clauses. Note that the two modes in the second row of Table 4 generate two different cases, *C3* and *C4*.

```

if
  □ Pressure = TooLow
    if
      □ Overridden = true -> SafetyInjection = Off  C1
      □ Overridden = false -> SafetyInjection = On  C2
    fi
  □ Pressure = Permitted -> SafetyInjection = Off  C3
  □ Pressure = High -> SafetyInjection = Off      C4
fi

```

Fig. 4. Function Defining SafetyInjection.

4.3 Branch Coverage

Associated with the method described in Section 4.2 is a precise, well-defined notion of test coverage. Our method assures branch coverage by observing the following rules:

1. In each condition table, every condition not equivalent to *false* is tested at least once.
2. In each event table, every event is tested at least once.
3. In each event table, in each mode, every no-change case is tested at least once.

For example, applying the first rule to the condition table in Table 4 generates a minimum of four test sequences, one each for the two conditions shown when `Pressure` is `TooLow` and one each for the two modes, `Permitted` and `High`, which guarantee that `SafetyInjection` is off.

The third rule, which addresses the no-change cases, is very important, especially for high assurance systems, where very high confidence is needed that a variable changes when it should change but does not change when it should not change. We found the coverage provided by the third rule too weak because many input events do not change any dependent variable. To achieve greater coverage, we modified our method so that it can generate test sequences satisfying a stronger rule, i.e.,

- 3'. In each event table, in each mode, a change in each monitored variable which does not change the value of the variable that the table defines is tested at least once.

For example, consider the no-change case *C2* for the function defining the mode class `Pressure` (see Figure 2). For this case, our method would generate three test sequences, one corresponding to a change in each of the monitored variables `Block`, `Reset`, and `WaterPres`. (Of course, `WaterPres` could only change to some other value below the constant `Low`.) Similarly, three test sequences would also be constructed for each of the other no-change cases, *C5* and *C7*. Hence, for this event table, our method would construct 13 test sequences.

We obtain coverage for the event table defining `Pressure` because the modes `TooLow`, `Permitted`, and `High` cover the state space. In many event tables, however, not all of the modes are mentioned explicitly because, in some modes, no event changes the value of the variable that the table defines. In such situations, the no-change cases that involve the missing modes are covered by appending an additional *else* clause to the function definition derived from the table.

5 A Tool for Automatically Generating Test Sequences

We have developed a tool in Java that uses a model checker to construct a suite of test sequences from an SCR requirements specification. To achieve this, the tool automatically translates the SCR specification into the language of either SMV or Spin, constructs the different cases, executes the model checker on each case and analyzes the results, derives the test sequences, and writes each test sequence into a file. For each case that it processes, the tool checks whether the case is already covered by one or more existing test sequences. If so, it proceeds to the next case. If not, the tool runs the model checker and transforms the counterexample generated by the model checker into a test sequence. As it processes cases, the tool sometimes finds that a new test sequence t_2 covers all cases associated with a previously computed test sequence t_1 . In this situation, the test sequence t_1 is discarded because it is no longer useful.

Because many software errors occur at data boundaries, we designed a tool option that causes extra test sequences to be constructed at data boundaries. Turning on this *comparison-split* option causes the tool to split a branch containing the relation $x \geq y$ (x and y are integers) into two branches, one containing $x > y$ and the other containing $x = y$. Similarly, this option splits a branch containing $x > y$ into two branches, one containing $x = y + 1$ and the second containing $x > y + 1$. When this option is selected, the tool will generate two test sequences to test the given relation rather than one alone.

Specif.	No. of Vars	No. of Branches	Total Test Seq.		Useful Test Seq.		Exec. Time		Total Steps	
			Spin	SMV	Spin	SMV	Spin	SMV	Spin	SMV
Small SIS	6	33	7	14	5	11	73s	3.7s	280	62
Large SIS	6	33	12	14	3	11	165s	4099s	10,529	778
Cruise Cont.	5	27	19	23	7	15	100s	5.1s	245	66
WCP1	55	50	15	-	10	-	500s	∞	4795	-

Table 5. Automatic Generation of Test Sequences Using Spin and SMV.

5.1 Experimental Results

This section describes the results of applying our tool to four specifications: the small SIS specification described in Section 4; a larger SIS specification; the small Cruise Control specification in [18]; and the WCP1 specification, a mathematically sound abstraction of a large contractor specification of a real system [12]. The larger SIS is identical to the small SIS, except Low is 900, Permit is 1000, and WaterPres ranges between 0 and 2000 and changes by no more than 10 psi per step. The purpose of applying our test generation tool to the WCP1 specification was to evaluate our method on a large, realistic specification. The WCP1 specification is a sizable component of a contractor specification in which five real-valued variables have been replaced by five integer variables.⁵ (Model checking the original contractor specification is infeasible because its state space is infinite.) The reduced specification is still quite large, containing 55 variables—20 monitored variables, 34 auxiliary variables, and one controlled variable. In processing all four specifications, the comparison-split option was off. In generating test sequences for the three smaller specifications, we applied rule 3' from Section 4.3 to obtain wider coverage. Due to the large size of the WCP1 specification and consequently the long execution time that we anticipated, in generating test sequences for WCP1, we applied rule 3, which is weaker.

Tables 5 and 6 summarize some results from our experiments. In Table 5, No. of Vars gives the total number of variables in each specification and No. of Branches the total number of branches, Total Test Seq. gives the total number of test sequences generated and Useful Test Seq. the number of test sequences remaining after weaker test sequences (those covered by other test sequences) are discarded, Exec. Time indicates the total seconds required to construct the test sequences, and Total Steps describes the total number of input events the tool processed in constructing the test sequences. For both Spin and SMV, Table 6 shows the number of “unreachable” cases the tool encountered (see below) and the lengths of the useful test sets generated for each specification.

5.2 Spin vs. SMV

Because their approaches to model checking are significantly different, Spin and SMV produced very different results, both in the test sequences generated and in their efficiency on large examples. As Table 6 shows, the main problem with Spin

⁵ Although the test sequences generated from the WCP1 specification contain abstract versions (i.e., discrete versions) of the real-valued variables, translating each abstract variable to one or more real-valued variables, while tedious, is straightforward. We have developed, and plan to implement, an algorithm that automatically transforms an abstract test sequence into a concrete test sequence.

Specif.	Spin-Generated Test Sequences			SMV-Generated Test Sequences		
	Useful	Unreach.	Lengths	Useful	Unreach.	Lengths
Small SIS	5	1	1, 20, 54, 99, 106	11	1	2, 3(2), 5(2), 6(2), 8(4)
Large SIS	3	1	3082, 3908, 3539	11	1	2, 3(2), 91(2), 101(4)
Cruise Cont.	7	7	31(2), 35, 37(4)	15	7	2(3), 3(3), 5(3), 6(6)
WCP1	10	2?	2(2), 3, 30, 72, 104 895, 1086, 1292, 1309	—	—	—

Table 6. Unreachable Cases and Test Sequence Lengths for Four Specifications.

is the length of the test sequences it generates. Because Spin does a depth-first search of the state-machine model, it produces very long counterexamples (for us, very long test sequences). Although we applied the Spin switch which finds the shortest counterexample, this approach rarely (if ever) found a test sequence with the shortest possible length.

This led us to experiment with SMV, which produces the shortest possible counterexamples because its search is breadth-first. To illustrate the results of generating test sequences using counterexample generation in SMV, we reconsider the branch labeled *C1* in Figure 2 (see Section 4.2). Table 7 shows the test sequence of length 3 that our tool derived from the counterexample generated by SMV. Clearly, this test sequence is a cheaper way to test a software implementation of SIS for the behavior described in case *C1* than the test sequence of length 20 shown in Table 3.

Step No.	Monitored Var. Value	Controlled Var. Value
1	WaterPres 5	
2	WaterPres 8	
3	WaterPres 10	SafetyInjection Off

Table 7. Test Sequence Derived from SMV Counterexample for Case *C1* of Fig. 2.

For each of the three specifications for which it produced results, using SMV to construct the test sequences dramatically reduced the length of the test sequences. However, SMV also produced many more test sequences than Spin. For example, in analyzing the smaller SIS specification (see Table 6), SMV produced 11 useful test sequences ranging in length from 2 to 8, whereas Spin generated five useful test sequences of lengths 1, 20, 54, 99, and 106.

As Tables 5 and 6 show, not only did SMV generate shorter counterexamples, in addition, for small examples, SMV was faster than Spin. However, for large examples, SMV required long computation times, whereas Spin was generally faster and covered the entire specification with fewer, but very long, test sequences. In the case of WCP1, SMV ran out of memory before it generated any test sequences (indicated by ∞ in Table 5). In contrast, Spin generated test sequences for every specification.

The reason for these difference lies in the different approaches to model checking taken by Spin and SMV. Spin uses explicit state enumeration to verify properties, i.e. computes the set of reachable states by enumeration (i.e. “running the model”), whereas SMV represents the reachable states symbolically as a BDD formula. Because the number of reachable states in requirements specifications is usually far fewer than the number of possible states and because the BDD

formula computed by SMV becomes enormous when it includes constraints on input variables, Spin often does better than SMV on large SCR specifications, especially in finding a state where the property is false [3].

5.3 “Unreachable” States

In generating test sequences for the three smaller specifications, our tool exposed several cases that involved “unreachable” states (see Table 6). For example, the tool found one unreachable state in each of the SIS specifications. In each, the model checker tried to find a trace in which the mode `Pressure` made a transition from `TooLow` to `High`. In both specifications, such a transition is impossible given the constraints on changes in `WaterPres` and the values of the constants `Low` and `Permit`. Similarly, all seven of the unreachable cases shown in Table 6 for the Cruise Control specification also involve impossible transitions; for example, a transition in which `IgnOn` changes and the mode class `CruiseControl` remains the same is easily shown to be impossible, using the invariants in [18].

For large specifications, a model checker sometimes runs out of memory (or time) before it finds a counterexample. In this situation, our method cannot detect whether the case is unreachable or is simply too complex to be analyzed by the model checker with the available memory. When this situation occurs, our tool identifies the case that the method failed to cover, so that the designer can consider the problem and, if necessary, build the test sequence by hand. This situation occurred when the tool was model checking the WCP1 specification with Spin and ran out of memory before it had generated test sequences for the two “unreachable” cases listed in Table 6. Our suspicion is that these two test sequences do not involve impossible transitions. Instead, the large size of the WCP1 specification probably caused Spin to run out of memory before it found traces for these two cases.

5.4 Role of Abstraction

Although the WCP1 specification has many more variables and is much more complicated than the larger SIS specification, Spin required many fewer input events in generating test sequences for the WCP1 specification than in generating test sequences for the larger SIS specification. In our view, the reason is that abstraction was applied effectively to the WCP1 specification; in contrast, no abstraction was applied to the SIS specification. In processing specifications, Spin usually changes *every* input variable in the specification, not only the “interesting” input variables, many times. By eliminating the uninteresting input variables using abstraction, it should be possible to decrease the length of the test sequences that Spin produces.

5.5 Effective Use of Model Checking

Although model checking may be used to *verify* properties of specifications, the enormous state space of finite-state models of practical software specifications often leads to the *state explosion problem*: the model checker runs out of memory or time before it can analyze the complete state space. This occurs even when partial order and other methods for reducing the state space are applied. Model checking is thus usually more effective in detecting errors and generating counterexamples than in verification [11, 3]. Hence, we are using a model checker in the most effective way.

6 Related work

At least two groups [22, 23] have formulated a formal framework where both testing criteria and test oracles are formally defined. In this approach programmers use the same model both to specify programs and to construct test sequences. While both groups present guidelines for constructing test sequences, unlike us, they do not describe a concrete method for automatically generating the test sequences.

Recently, three other groups have used a model checker to generate test cases complete with output values [5, 8, 1]. Callahan and his colleagues use a process representing the specification to examine traces generated by a process simulating the program. In this manner, they detect and analyze discrepancies between a software implementation and the specification (a set of properties). They use Spin as an oracle to compute the system outputs from the process representing the specification. Engels et al. also describe the use of Spin to generate test sequences [8]. They assume the designer has defined the “testing purpose,” analogous to our trap properties. A major weakness is the reliance of their method on a manual translation of the specification to Spin, which requires some skill and ingenuity. Because both methods use properties to construct the test sequences, they suffer the weaknesses of property-based methods described in Section 4.1. Ammann, Black, and Majurski have proposed a novel approach based on mutations, which uses SMV to generate test sequences [1]. By applying mutations to both the specification and the properties, they obtain a large set of test sequences, some of which describe correct system executions and others describing incorrect system executions. Using this method, a correct software implementation should pass tests that describe correct executions and fail tests for incorrect executions. However, this method lacks the systematic treatment of the no-change cases provided by our method.

Blackburn et al. [4] describe a different method for generating test sequences from SCR specifications, which does not use a model checker. In their method, a tool called T-VEC derives a set of test vectors from SCR specifications. In this vector-oriented approach, each test sequence is simply a prestate/poststate pair of system inputs and outputs. Although this method has proven useful for testing software modules, its use in black-box software testing is problematic, because it does not provide a valid sequence of inputs leading to each pair of state vectors.

7 Summary and Plans

This paper has described an original method for automatic generation of test sequences from an operational requirements specification using a model checker. The method has several desirable features:

- It uses an operational requirements specification both to construct a valid sequence of system inputs and to compute the expected system outputs from the input sequence.
- It generates a suite of test sequences that cover the state transitions by generating test sequences from cases explicitly described in the specification *and* from cases that are implicit in the specification (the “no-change” cases). These sequences test the most critical input sequences, those that should change the system state (as specified in the event tables) and those that

should not change the system state. Every condition in a condition table is also tested.

- It may be applied using either the explicit state model checker Spin or the symbolic model checker SMV.

To illustrate the utility of our approach, we showed how a tool that implements our method can generate test sequences for some small examples and for a large component of a contractor specification of a real-world system. These early results demonstrate both the method's potential efficiency and its practical utility.

A number of other important issues remain. First, we plan to experiment with abstraction to address both the state explosion problem encountered with SMV and the long input sequences produced by Spin. Given the effectiveness of combining mathematically sound, automated abstraction methods with model checking for detecting specification errors [12], we are optimistic that abstraction can also prove effective in making automatic test sequence generation from large specifications practical. Second, we will study alternative methods for selecting test sequences for a given branch. Our current method usually constructs a single test sequence for each branch of a function definition. An important question is how to select a collection of test sequences that are adequate for testing the behavior specified by that branch. One alternative which could prove useful when a large number of variable values exist (e.g., large ranges of numerical values) is to use a statistical method to select test sequences. Another alternative is to select test sequences systematically by further case splitting. To determine which particular test sequences to select, a method like that of Weyuker and her colleagues [24] may be useful. Finally, we plan to use the suite of test sequences that our tool generates from a given SCR requirements specification to test a real software implementation.

Acknowledgments

We are grateful to M. Archer, R. Jeffords, D. Mandrioli, and the anonymous referees for their constructive comments on earlier versions of this paper.

References

1. P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE Intern. Conf. on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December 1998.
2. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.
3. R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Eng. J.*, 6(1), January 1999.
4. M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.
5. J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proc. SPIN Workshop*, Rutgers University, August 1996. Tech. Report NASA-IVV-96-022.
6. P.-J. Courtois and David L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, Baltimore, MD, 1993.

7. S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 1997.
8. A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proc. TACAS'97*, pages 384–398. Springer, 1997. in E. Brinksma, editor, LNCS 1217.
9. 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.
10. S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. on Softw. Eng.*, 17(6), June 1991.
11. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proc. Formal Methods Europe (FME'96)*, pages 662–681. Springer-Verlag, March 1996. LNCS 1051.
12. 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), November 1998.
13. 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.
14. 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.
15. K. Heninger, D. Parnas, J. Shore, and J. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.
16. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
17. W. E. Howden. A functional approach to program testing and analysis. *IEEE Trans. on Softw. Eng.*, 15:997–1005, October 1986.
18. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, November 1998.
19. D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. on Computer Systems*, 13(4):365–398, 1995.
20. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Englewood Cliffs, NJ, 1993.
21. S. P. 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.
22. D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th Intern. Conf. on Software Eng.*, pages 105–118. Springer, May 1992.
23. P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Trans. on Softw. Eng.*, 22(11):777–793, November 1996.
24. E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Trans. on Softw. Eng.*, 20:353–363, May 1994.