Alloy and SCR: An Evaluation and Comparison

Justin R. Erenkrantz, Scott Hendrickson

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{jerenkra,shendric}@ics.uci.edu

Abstract

Alloy and SCRtool are two popular toolsets in the area of formal methods. Therefore, it may be useful to evaluate these tools and compare them in order to obtain a clearer picture of how formal methods can be used in practice. Alloy is a constraint checker built upon a strong mathematical foundation. However, use of Alloy involves a steep learning curve and the payoff is delayed. It is an especially useful tool when the project is expected to evolve. SCRtool provides the required tools to completely specify the external behavior of a system. However, SCRtool could be improved in practice by encouraging the proliferation of the toolset publicly, the creation of an unified SCR semantic, and an improved user interface. Additionally, we conclude, based upon our evaluation, that a mapping from SCRtool to Alloy is feasible, while the mapping from Alloy to SCRtool is a current open research question.

1. Introduction

According to one source, "formal methods is the applied mathematics of computer system engineering."[14] More concretely, formal methods can be used to create a model of a system in order to understand how the system will behave in the real world. Due to the origins in mathematics and logic of formal methods, these areas lend themselves to automated tools that assist the developer in this discovery process.

Without tools, the process of validating and verifying these requirements would be intractable. Even the smallest example can pose difficulties in manually verifying and validating the requirement without relying upon strong formalisms. This challenge only increases as the system approaches the scope of real-world systems. Therefore, in order to appropriately manage these specifications, an automated toolset will be required.

Alloy and SCRtool are two popular toolsets in this area of formal methods. However, as we will discuss in this paper, they take different approaches to helping apply these formal methods. Alloy is primarily a constraint checker that validates that a specification is built upon a solid logical foundation. In contrast, SCR focuses on defining the external behavioral specifications of a system in order to understand how the system will react to events and ensure that erroneous states are not encountered.

For the remainder of this paper, we will proceed along parallel tracks discussing Alloy and SCR. We start by providing the necessary background for both Alloy and SCR in order to understand what each tool can achieve. We will then continue with an overview of the technical features of each tool. Next, we will examine the functionality of each tool in order to understand how the tool can aid in the creation and verification of formal specifications. After this examination, we will work through examples with each tool to demonstrate what the tool can do.

Subsequently, we will then provide some observations that we have arrived at after our exposure to these tools. Then, we will perform a comparison between Alloy and SCR. In this section, our main goal will be to differentiate these tools and place them in their appropriate context within both the software engineering and formal methods paradigm. In conclusion, we will discuss some future work that may be needed in order to conduct better comparisons between these tools.

2. Alloy

Alloy is a tool that models and analyzes systems. It was developed by MIT's Software Design Group and formally introduced in [12]. Alloy is intended to be used on a micromodel of a system, which is one that is orders of magnitude smaller than the actual system. There are two primary components to Alloy: the Alloy language and the Alloy Analyzer.

The Alloy language [10] is a small modeling language that can be used to express the basic structure of a model as well as constraints and operations specifying how that structure may change. It builds upon boolean algebra, set theory, quantifiers, and first-order relational logic. Analysis of a specification written using the Alloy language can be automated, which differentiates it from its predecessors.

The starting point for the Alloy language was Z [17]. Alloy addresses the following deficiencies of Z:

1. *that it does not lend itself to automation*
2. *that it is incompatible with object modeling idioms, and*
3. *that it is dependent on LaTeX.*

The Alloy Analyzer, previously called Alcoa [11], analyzes a specification model written in the Alloy language. Since the Alloy language is undecidable, it is not possible to provide a sound and complete analysis of a specification. Instead, the Alloy Analyzer finds models of formulas, that is assignments of values, that make the formula true or that contradict the formula within a given scope. These formula are expressed as examples or counter-examples of a system.

A system with constraints that are too weak will have a model that meets the constraints, but violates its assertions. Such a model is referred to as a counter-example. A system where the constraints are too strong (meaning that the constraints contradict each other) will have no model that satisfies all the constraints since the model would have to contradict itself. For such a system, Alloy tries to find a model, if it does, then the model is referred to as an example.

It is important to note that when Alloy does not find a counter-example demonstrating weak constraints in a given scope, it does not mean that a counter-example will not exist in a larger scope. However, in practice, if a counter-example exists, it can typically be found in a small scope, which indicates that the user can have a reasonable degree of assurance about a system.

2.1. Language overview

The Alloy language uses a combination of boolean algebra, set theory, quantifiers, and first-order relational logic. The language expresses statements using ASCII characters. Figure 1 shows an example Alloy specification from the Alloy reference manual, which shows a specification's basic structure.[13]

```
-- first Alloy example
module CeilingsAndFloors
sig Platform {}
sig Man {ceiling, floor: Platform}
fact {all m: Man | some n: Man | Above (n,m)}
fun Above (m, n: Man) {m.floor = n.ceiling}
assert BelowToo
       {all m: Man | some n: Man | Above (m,n)}
run Above for 2
check BelowToo for 2
```

Figure 1: An example Alloy specification

It is important to discuss the concepts behind the Alloy language before we discuss its syntax and expressiveness.

Atoms and Relationships

An atom is a typed entity that is indivisible, immutable, and uninterpreted. This means that an atom cannot be broken into smaller parts, that it doesn't change, and that it doesn't have any built-in meaning associated with it. Alloy uses atoms to represent something in the real world or a property about a system. Two atoms are equal if they are the same atom.

Relationships group atoms into tuples, which are ordered sequences of atoms. Relationships are also typed in that each atom in a tuple must be of a certain type. A relationship type may specify one or more atom types, but not zero. Relationships may then contain one or more tuples of atoms of the specified types, or no tuples at all, in which case the relationship is said to be empty. Relationships are equal if they contain the same set of tuples (which of course contain the same atoms).

It is important to note that Alloy does not differentiate between scalars (an atom), tuples (an ordered set of atoms), and sets of tuples (a relationship). They are all treated the same. Therefore, an atom, A, a tuple containing only that atom, (A), and a set containing that tuple, {(A)}, are all considered to be identical.

For purposes of this discussion, we can consider a relation to be a table where each row is a tuple, and each column corresponds to the i^{th} element in each tuple. This would mean that the order of rows does not matter, since each row represents a tuple in the set of tuples that makes up the relationship. However, the order of columns does matter because the order of atoms in each tuple is significant.

It is possible to see that a relationship can represent many different types of associations between atoms. For example, a function may be expressed as a relationship by considering the first column to be the output of a function, and the other columns as the input to that function. A set can be considered as a relationship with only one column.

We now discuss how useful information about relationships and atoms is expressed in Alloy. This includes, expressing types, constraints, facts, and first-order relational logic.

Signatures

Signatures consists of a basic type and a set of atoms. Signatures are declared using the keyword *sig*. A signature creates an implied type that can not be referred to explicitly. The following statement creates a signature of type 'Person':

```
sig Person{}
```

Signatures may also declare fields. The following statement creates a signature of type *Person* with two fields *spouse* and *parents*.

```
sig Person{spouse: option Person, parents: set Person}
```

The above signature also creates a *spouse* relationship mapping each *Person* to another *Person*. Notice that the *spouse* field may be empty since it is optional. The second relationship, *parents*, maps each *Person* to a set of other *Persons*. As it is currently stated, a person may have no parents (when the *parents* set is empty), or multiple parents (when the *parents* set is non-empty). Of course, this doesn't make sense, and needs additional restrictions that limit the set of *parents* to only two people. This can be accomplished using *facts*, which we will discuss facts shortly.

Given a scalar *p* in *Person*, we can dereference *p*'s *spouse* with the expression *p.spouse*. If *p* is married, then *spouse* will contain another scalar person in *Person*. Otherwise, *p* will contain the empty set. Since Alloy treats scalars and sets identically, you can think of *p.spouse* as returning a subset of *Person*, which may be empty or contain only one person. In a similar fashion, the expression *p.parents* will return a subset of *Person*. However, *parents* may be a set of multiple persons.

Sets may also be subdivided into smaller sets. The following statement subdivides the set of all *Person* into two subsets: *Man*, and *Woman*.

```
part sig Man, Woman extends Person {}
```

This statement means that there are now two sets called *Man* and *Woman*. Each scalar element in the set *Man* or *Woman* is also an element in *Person*. Additionally, each element in *Man* and *Woman* also has the fields that are included in *Person*.

As already mentioned, for our model to make sense we need to add additional restrictions to it. Alloy provides a means of doing this using facts.

Facts

A fact creates a restriction on relationships that limit the possible values that they may contain. The *Parents* signature adds a number of implicit facts, which we have already discussed:

- each scalar in *spouse* is in *Person*
- *spouse* may be empty or have only one element
- *parents* is a subset of *Person*

Granted, these three implied facts are not very interesting, yet they are important to note. It would make more sense for our model if we add additional facts, specified using the *fact* operator. One fact we might add would be:

```
fact { Man.spouse in Woman
       && Woman.spouse in Man }
```

This fact states that the *spouse* of all elements in the *Man* set must be in the *Woman* set and vice versa. The double ampersand is a logical operator meaning conjunction (we will discuss these later). A fact should always evaluate to true.

Assertions

Assertions are statements that should be true about the system. They serve as checks to make sure that the system is behaving correctly. These differ from facts because the Alloy Analyzer uses these to validate states of the system rather than restrict the possible states of a system. A well behaved system should never contradict an assertion and should behave the same regardless of whether the assertions are present. An example assertion might be:

```
assert ChildrenHaveParents { all p: Person | p.parents }
```

The assertion above states that all *Persons* have a non-empty set of *parents*. A counter-example for this system would have at least one person who did not have any parents.

Functions

A function is a reusable formula that can be applied to a set of typed parameters. In Alloy, convention dictates that the second parameter acts as a return value. However, multiple parameters can be used as return values if so desired. Functions in Alloy are similar to functions in functional programming languages. However, in Alloy, they are used to transition from one state to another. The following function is taken from another example provided with Alloy:

```
fun BusyDay (bb: BirthdayBook, d: Date){
   some cards: set Name |
      Remind (bb,d,cards) && !sole cards
}
```

The function above applies to a birthday book example where a birthday book contains a list of names and birthdates. The function BusyDay determines if there are multiple names with a birthday on a given date.

Table 1: Alloy Logic Operators

Logic Operators	Math Symbol	ASCII
Disjunction	$\alpha \vee \beta$	a || b
Conjunction	$\alpha \wedge \beta$	a && b
Negation	$\neg\alpha$	!a
Implication	$\alpha \rightarrow \beta$	a => b
Bi-implication	$\alpha \leftrightarrow \beta$	a <=> b

Table 2: Alloy Quantifier Operators

Quantifier Operators	Math Symbol	ASCII
Universal	$\forall\alpha$	all a
Existential	$\exists\alpha$	some a
Non-existential	$\neg(\exists\alpha)$	no a

Table 3: Alloy Set Operators

Set Operators	Math Symbol	ASCII
Union	$\alpha \cup \beta$	a + b
Intersection	$\alpha \cap \beta$	a & b
Difference	$\alpha - \beta$	a - b
Membership	$\alpha \in \beta$	a in b

Operators

Many of the operators in Alloy are very closely related to first-order relational logic. There is generally a direct mapping from first-order relational logic to the Alloy language. We discuss the operators here and present their ASCII notational equivalents.

There are three categories of operators in Alloy: Standard logical operators, quantifiers, and set operators. The logical operators supported include disjunction, conjunction, negation, implication, and biimplication and are a part of boolean algebra. Quantifiers that are supported include universal, and existential among others. These come from first-order relational logic. Standard set operators such as union, intersection, difference, and membership are supported as well. These come from set theory.

Modules

Finally, Alloy uses a simple module system to split specifications into smaller, more manageable, and reusable pieces. This allows limited separation of concerns. A specification may include the text or concepts from other files by including the appropriate modules. Or, a specification may be self contained. Module names are used for scope resolution.

2.2. Tool Functionality

The Alloy Analyzer is a GUI application that has three main sections. One section is a rudimentary editor for modifying Alloy specifications. Another section displays detailed information about any solution (the example or the counter-example) that the tool finds. The last indicates information pertaining to the internal data structures used by the tool during analysis.

The tool provides a means for editing, building, and testing specifications. First, the tool provides the basic functions of a specification editor by allowing the user to load, edit, and save specifications and their modules. While a specification is loaded, the tool will compile the specification and report back to the user certain compile time errors. In this way, it helps a user to write well-formed specifications. The tool will then find any commands specified in the specification and present them to the user for execution.

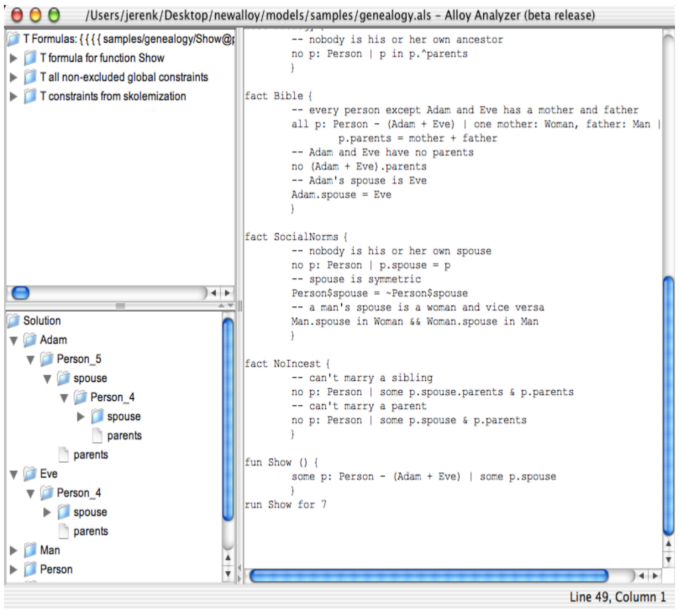
The Alloy Analyser will list the commands found in the specification that will initiate a search. There are two commands: *check* and *run*. The *check* command is used to instruct the Alloy Analyzer to find a counter-example. It validates the assertions to make sure that they hold given the constraints of a system. The *run* command instructs the Alloy Analyzer to find an example satisfying all constraints of the system. Both commands also specify the scope of the search space by placing an upper bound on the number of atoms and types of atoms used in the search space. These commands may be embedded in the specification.

The Alloy Analyzer is also capable of presenting a solution in a graphical format. The graphical representation consists of nodes, which represent atoms, and labeled lines, which represent relationships between those atoms. It is possible to customize the graph by selecting colors and shapes for each node and by specifying the color and font of each label, or even whether that label is included in the graph. In version 2.1, the Alloy Analyzer integrates the graph into the main user interface.

2.3. Technical Details

The Alloy Analyzer is available in three different formats. A graphical user interface (GUI) version, written in Java, is available as a stand alone application that uses native code for finding solutions. Version 2.0 is available for most platforms. Version 2.1 is only available as a beta for the Windows and Max OS X operating systems. The Alloy Analyzer is also available as a command-line tool and as an API for use in other tools.

A number of examples are included in the distribution of Alloy. The examples range from very simple to very complex. Examples include published systems such as the FireWire protocol, puzzles such as the Towers of Hanoi, and systems such as a File System to name a few. We will discuss the genealogy example and touch on the birthday example in this paper.


/Users/jerenk/Desktop/newalloy/models/samples/genealogy.als – Alloy Analyzer (beta release)
T Formulas: {{{{samples/genealogy/Show@
T formula for function Show
T all non-excluded global constraints
T constraints from skolemization
Solution
Adam
Person_5
spouse
Person_4
spouse
parents
parents
Eve
Person_4
spouse
parents
Man
Person
Line 49, Column 1

```
        -- nobody is his or her own ancestor
        no p: Person | p in p.^parents
        }

fact Bible {
        -- every person except Adam and Eve has a mother and father
        all p: Person - (Adam + Eve) | one mother: Woman, father: Man |
                p.parents = mother + father
        -- Adam and Eve have no parents
        no (Adam + Eve).parents
        -- Adam's spouse is Eve
        Adam.spouse = Eve
        }

fact SocialNorms {
        -- nobody is his or her own spouse
        no p: Person | p.spouse = p
        -- spouse is symmetric
        Person$spouse = ~Person$spouse
        -- a man's spouse is a woman and vice versa
        Man.spouse in Woman && Woman.spouse in Man
        }

fact NoIncest {
        -- can't marry a sibling
        no p: Person | some p.spouse.parents & p.parents
        -- can't marry a parent
        no p: Person | some p.spouse & p.parents
        }

fun Show () {
        some p: Person - (Adam + Eve) | some p.spouse
        }
run Show for 7
```

Figure 2: Screenshot of Alloy Analyzer User Interface

2.4. Examples

The first example we will use for Alloy is a toy model of genealogical relationships. It was originally written by Daniel Jackson and is distributed with Alloy. We will now highlight the important parts of this example.

```
sig Person {spouse: option Person, parents: set Person}
part sig Man, Woman extends Person {}
static sig Eve extends Woman {}
static sig Adam extends Man {}
```

This declares a set *Person*, with an optional field *spouse*, and a set of *parents*. The set of all people is split into a *Man* and a *Woman* subsets. It also defines a *Man*, called *Adam*, and a *Woman*, called *Eve*. The following facts are declared about the system:

```
no p: Person | p in p.^parents
```

This fact states that no person is their own parent. The caret operator means recursively include all parents' parents. Therefore, this actually says that no person is their

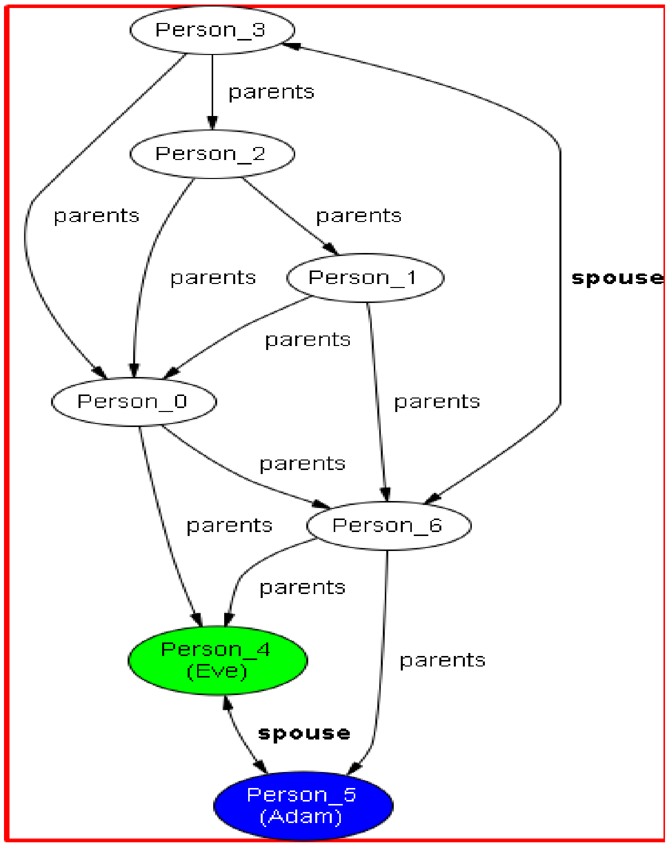


Figure 3: Example of Geneology

own ancestor.

```
all p: Person - (Adam + Eve) |
one mother: Woman, father: Man |
p.parents = mother + father
```

This says that all people except *Adam* and *Eve* have a *mother*, who is a *Woman*, and a *father*, who is a *Man*, and that together they make up their parents. Reading this statement, you might assume that it explicitly states that *Adam* and *Eve* do not have parents. However, this is not so, it says nothing about *Adam* or *Eve*.

```
no (Adam + Eve).parents
```

This fact explicitly states that *Adam* and *Eve* have no parents by stating that the parents set must be empty.

```
Adam.spouse = Eve
```

Finally, *Adam* and *Eve* are declared to be married. The next statements have to do with people in general and are not specialized to *Adam* and *Eve*.

```
no p: Person | p.spouse = p
Person$spouse = ~Person$spouse
Man.spouse in Woman && Woman.spouse in Man
```

The first statement above states that no person is their own spouse. The second line states that two people are each other's spouse. The last line states that the spouse of a *Person* in the *Man* set must be in the *Woman* set and vice versa.

```
no p: Person | some p.spouse.parents & p.parents
no p: Person | some p.spouse & p.parents
```

These two statements restrict people from marrying a sibling or a parent. The first statement claims that one person's parents and their spouses parents cannot have any common people. If they did, then the two people would be siblings. The second statement claims that a person's spouse and their parents should not contain the same people. Otherwise they would be marrying one of their parents. The function above states that there should be at least one person (other than *Adam* and *Eve*) that has a spouse.

```
fun Show () {
    some p: Person - (Adam + Eve) | some p.spouse
}
```

A command is also included in the specification. The command, shown below, instructs Alloy to search for an example that satisfies all the constraints using no more than seven instances of each type. Runnin Alloy on the following command produces the results in Figure 3.

```
run Show for 7
```

As we can see in the results shown from executing the above command, Alloy found a model that satisfies all constraints using a maximum of seven instances of any type. We also see that while we have made rules about who can marry who, we did not make any statements about who could have children. So, while there is no official incest due to marriage, in the model there is unofficial insest according to lineage. It is also easy to see that the model meets the constraints of the function. Two other persons, besides *Adam* and *Eve*, are married: *Person_6*, and *Person_3*.

We turn to another example specification to see how Alloy search for counter-examples. In this specification Alloy models a birthday book. The birthday book keeps track of a list of people and their birthdays. The following assertion claims that whenever a name is added to the birthday book and then removed, the birthday book should remain unchanged. .

```
assert DelIsUndo {
  all bb1,bb2,bb3: BirthdayBook, n: Name, d, d': Date|
    AddBirthday (bb1,bb2,n,d) &&
    DelBirthday (bb2,bb3,n)
    => bb1.date = bb3.date
}
```

The above assertion fails. Adding a name to the birthday book and then removing that name can result in a list of names that remains altered. The graph of the counter-example in Figure 4 shows that the problem occurs when the birthday book already contains the name that was added. In

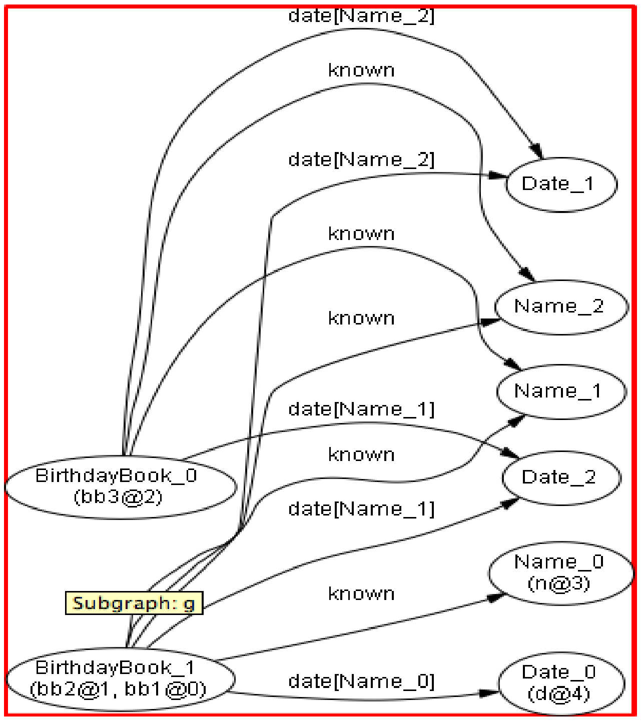


Figure 4: Counterexample of DelIsUndo

such a case, the old name is simply replaced with a new entry by the same name. When the name is removed, the new entry is removed, but the old entry by the same name is not restored. Therefore, this assertion fails in such a case.

Note that the example does not refer to one birthday book, but rather it refers to three. This is because the functions used in the assertion always return modified copies of the original birthday books rather than modifying the birthday book itself. In this example, *bb1* and *bb3* should be the same while *bb2* should have the new entry in it.

3. SCR

Software Cost Reduction (SCR) is a technique for describing the external behavior of a system and how it will react to these events. SCR was first developed in the mid-to-late 1970s at the Naval Research Laboratory and the Naval Weapons Center in order to update the flight program for the A-7 aircraft[7]. This project was selected as a case study because the SCR designers were hoping to demonstrate that the overhead incurred by the process would not be detrimental to either the time to development or runtime efficiency of the project[7].

The project leaders in [7] came up with six objectives that they wanted their technique to be able to achieve:

1. Specify external behavior
2. Specify constraints on the implementation
3. Be easy to change
4. Serve as a reference tool
5. Record forethought about the life cycle
6. Characterize acceptable responses to undesired events

Over time, the SCR technique has been refined by the Naval Research Laboratory. Additional revisions to the flight software of the A-7E aircraft have also been conducted using SCR.[1] As an extension of the original SCR work, Parnas has introduced the four variable method that defines variables as monitored, controlled, input, or output.[15] Projects such as SC(R)* and SC(R)^3 have been introduced to promote SCR by general practictioners in software engineering.[5,Chechik, 1998 #2203]

The distinguishing characteristic of SCR that separates it from other techniques is that its key concepts may often be represented in a tabular format. By representing the requirements as a table, it allows a visual representation that may be easier for humans to understand and also allows itself to be analyzed using formal methods.

3.1. Language

There are many conceptual aspects to the SCR language. SCR separates these aspects into three main classes: types, variables, and functions. Each of these play a specific role in the SCR specification process and will be described in this section.

Types

A variable is associated with a type property which indicates the legitimate values that it can represent. These types can be referenced by variables. Types may be enumerations with their legal values defined. They may also be defined as integers or floats with a constrained range.

Variables

SCR supports several different classes of variables: monitored, term, controlled. Each variable is associated with a type that constrains their range. Furthermore, each variable may have an initial value specified that will determine its state when the system is initialized.

Monitored. A monitored variable is used in SCR to indicate that the variable may change in response to environmental conditions and events. These variables are primarily utilized in mode transition tables.

Term. A term variable is used as a shorthand to reference a combination of monitored or controlled variables. Term variables are fully described by their associated condition function.

Controlled. A controlled variable is utilized in conjunction with event functions. A controlled variable is adjusted based upon events that occur in the system. Controlled variables are described by the relevant event function.

Assertions

Assertions are used in SCR to indicate conditions that the system should always satisfy. When the system is evaluated for correctness, all possible states should have these assertions satisfied. If they are not satisfied, then the system is deemed to be inconsistent.

Assumptions

SCR's assumptions are external conditions that are imposed by the environment. In a sense, these assumptions are axioms that govern the nature of the system. These assumptions are always true. If these are violated, then something in the environment has been altered.

Modeclass

A modeclass defines the possible legal states that a system may be in. These modes are used in the events, conditions, and the mode transition tables.

Conditions

A condition is related to a term variable in that it defines the expression that governs the value of the term variable. A condition function may either be associated with modes or be modeless. When a condition function is modal, the term variable may have its definition altered depending upon which mode the system is in. When the condition function is modeless, the term variable will maintain the same definition regardless of the system state.

These condition functions should be disjoint and covered. This means that all possible values of the term variable are defined and there is no overlap in the values based upon the expression. We will discuss the exact meaning of disjointness and coverage in relationship to SCRtool later in this section.

Table 4shows an example of a condition function table. The modes are located in the first column, with the conditions comprising the rest of the columns in the table. If the condition function is modeless, then only one row is present. Otherwise, if the condition functional is modal, there will be a row for each mode that the system can enter. The bottom row contains the term variable name that is controlled by this function in the first column of this last row, and the subsequent columns in the last row enumerates all potential values of this variable according to its type.

Table 4: Condition Function Example

Modes	Conditions	
-	Cond1 = True AND Cond2 = False	Cond1 = False AND Cond2 = True
Ready =	TRUE	FALSE

Events

An event function describes how changes in the system affect controlled variables. The structure of the event function table is similar to the condition table (see above). However, instead of having expressions that govern the change of the variables, the values are changed when a monitored variable changes value.

Mode Transition Table

A mode transition table describes how a system responds to events by modifying its internal state in response to changes in monitored variables. Each row in the table describes one atomic transition between system states. An example of a mode transition table is provided in Table 5, which is further explained in Section 3.4 along with its relevant background information to place it in its appropriate context.

Mode transition tables are linked to a single modeclass which enumerates the possible legal states that the system may be in at any particular time. According to the SCR semantics, a system may only be in exactly one mode from a modeclass at a time. Therefore, it is not possible for a system to be in two modes in the same modeclass at the same time. Nor can the system not be in one of the listed modes as it must be in some determined state. Each mode transition table has a defined initial state that determines what its state will be when the system is brought up.

For each row in a mode transition table, the first column indicates which mode the system must be in before the particular transition can be activated. The last column indicates what the resulting mode of the state will be. The intermediate columns depict the input to the mode transition table - listing either the dependencies that must be satisfied or the event that must occur to satisfy this transition.

These intermediate columns may either be a precondition to the transition, or be the event that triggers the event. A precondition is denoted by a truth value that represents an expression. If a particular cell is false (denoted by the lowercase letter f), then it means that the expression in that column must be false before the transition can occur. Likewise, if the cell value is true (denoted by the lowercase letter t), then the expression must be true before the condition is true.

However, if the value in the cell is @T or @F, it means that this row is evaluated when that expression becomes either true or false. If the cell value is @T, this means that the transition will only fire when in the previous state the expression was false, then in the current state, the expression is now true. If an expression does not change relative to its prior state, then any @-expressions are not evaluated.

While a transition may have multiple preconditions, it may only have one event that causes the condition to be triggered. If all of the preconditions are met and the cell value changes to the specified @T or @F, then that row is then evaluated and the mode is adjusted accordingly to the last column in that row.

3.2. Technical Details

The package that was evaluated was SCRtool and SCRsimulator from Naval Research Laboratories. It is currently at version 2.1.0 in targeted beta releases. SCRtool is an extension and rewrite of the older SCR* toolset that builds upon the experiences of the prior version.[5]

The implementation of SCRtool is primarily Java-based allowing for maximum portability. However, the SCR testtool backend is written in C++ and requires native binaries. Consequently, only Windows and Mac OS X binaries are distributed by the NRL. SCRtool may also take advantage of SPIN if it is available, but SPIN is distributed separately from SCRtool and SCRsimulator.

The package is split up into two programs: SCRtool and SCRsimulator. SCRtool is responsible for editing the specification and performing basic validation of the specification. SCRsimulator can model a specification created in SCRtool and visualize it in a manner which may make it easy for people to interact with the proposed model.

3.3. Tool Functionality

Most of the functionality of the toolset resides in SCRtool. Keeping with the tradition in SCR in relying upon tables, SCRtool's main user paradigm is the tabular user interface. Since most SCR practictioners are already assumed to be comfortable with tables, it relies heavily on this motif in order to convey key concepts to the user.

Syntax Tree

As depicted in Figure 5, the syntax tree is the principal user interface in SCRtool. The syntax tree visually separates the specification into dictionaries and functions. The dictionaries are further sub-divided by primary classification (type, variable, assertion, etc.). The dictionaries and how they are incorporated into SCRtool are examined in a later section.

Each entry in the syntax tree displays its name. If the entry is related to a table or dictionary, a link to the appropriate table window is included. Additionally, a link to the dependency graph is available that includes that node. If an error is currently associated with the entry, the name will appear in red. An explanation of the error will then be present underneath the name. If a counterexample is found that disproves the claim, the state will be detailed along with this description.

Dictionaries

As shown in Figure 6, the interface for adding or modifying variables to an SCR specification via SCRtool is via the dictionary interface. Through this interface, all monitored, term, and controlled variables are displayed together. By utilizing the search functionality, specific variable names may be looked up. Each variable receives its own row in the dictionary, which details the name, class, and type of the variable.

The user may also indicate what the initial value and expected accuracy of this variable will be. If the variable is a term or controlled variable, the dictionary links with the associated function that governs the variable. The user may also specify in a free-form textual format how the value should be interepreted by the system. This may be beneficial in annotating the variable with its meaning and other notes that may assist collaborators in understanding.

Function tables

SCRtool allows you to define functions utilizing the table paradigm prevalent in SCR. However, there are some subtle differences between the common SCR representation of these tables and how SCRtool uses them.

A shorthand notation used in SCRtool for the mode transitions represented in a single row is to use the WHEN keyword. Rather than represent the mode transitions as a table with columns for each possible expression used in that table, SCRtool utilizes an expression to summarize each of the intermediary columns.

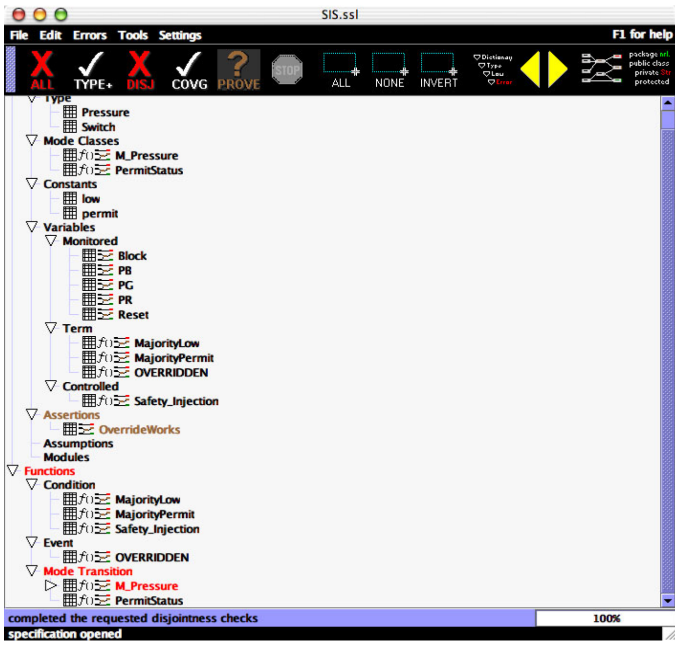
For example, the following row in Table 5:

	Ignited	Running	TooFast	Brake	Activate	Deactivate	Resume	New Mode
Inactive	t	t	-	f	@T	-	-	Cruise

would be represented as:

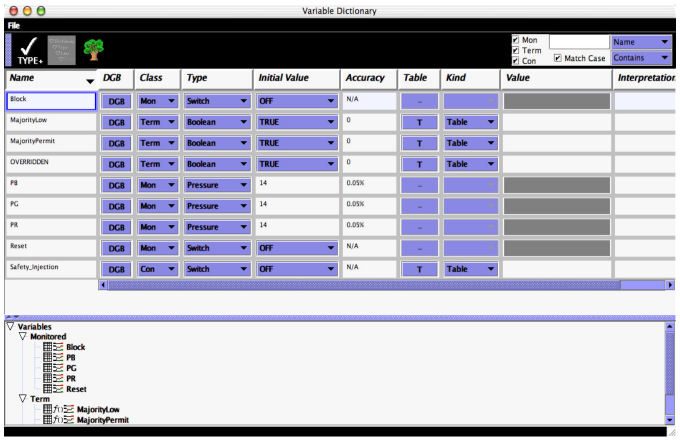
@T(Activate) WHEN
(Ignited AND Running AND NOT Brake)

This format used by SCRtool trades off expressiveness


SIS.ssl
File Edit Errors Tools Settings F1 for help
ALL TYPE+ DISJ COVG PROVE STOP ALL NONE INVERT
Dictionary Type Lau Error
package nrl. public class private Str protected

```
  ▽ Type
      Pressure
      Switch
  ▽ Mode Classes
      M_Pressure
      PermitStatus
  ▽ Constants
      low
      permit
  ▽ Variables
    ▽ Monitored
        Block
        PB
        PG
        PR
        Reset
    ▽ Term
        MajorityLow
        MajorityPermit
        OVERRIDDEN
    ▽ Controlled
        Safety_Injection
  ▽ Assertions
      OverrideWorks
    Assumptions
    Modules
▽ Functions
  ▽ Condition
      MajorityLow
      MajorityPermit
      Safety_Injection
  ▽ Event
      OVERRIDDEN
  ▽ Mode Transition
    ▷ M_Pressure
      PermitStatus
completed the requested disjointness checks    100%
specification opened
```

Figure 5: Syntax Tree in SCRtool



Name	DGB	Class	Type	Initial Value	Accuracy	Table	Kind	Value	Interpretation
Block	DGB	Mon	Switch	OFF	N/A	–			
MajorityLow	DGB	Term	Boolean	TRUE	0	T	Table		
MajorityPermit	DGB	Term	Boolean	TRUE	0	T	Table		
OVERRIDDEN	DGB	Term	Boolean	TRUE	0	T	Table		
PB	DGB	Mon	Pressure	14	0.05%	–			
PG	DGB	Mon	Pressure	14	0.05%	–			
PR	DGB	Mon	Pressure	14	0.05%	–			
Reset	DGB	Mon	Switch	OFF	N/A	–			
Safety_Injection	DGB	Con	Switch	OFF	N/A	T	Table		

```
▽ Variables
  ▽ Monitored
      Block
      PB
      PG
      PR
      Reset
  ▽ Term
      MajorityLow
      MajorityPermit
```

Figure 6: Dictionary in SCRtool

for the rigid format of the table. Using the straight tabular format of SCR, it is not possible to directly create complicated expressions for use in a mode transition table. To solve this problem, term and controlled variables can be used to indirectly refer to the expression. However, with SCRtool, these term and controlled variables are not required as they can be directly expanded into the mode transition table expression.

Dependency Graph

One of the features that SCRtool provides is displaying a dependency graph of all of the elements within the specification. The graph shows the relationship between the elements. For example, it visually indicates the types that specific variables are derived from. It also depicts what variables are used by the function. This allows a perspective of where an element fits into the overall specification. Therefore, if a change is required, it is possible to visually recognize what the effects of this change may be.

Automated Checking

SCRtool can perform syntax, disjointness, and coverage checks on the current specification. These inform the user about any current inconsistencies in the specification as it is being edited, so that the user can resolve them immediately. Initially, these checks are displayed on the toolbar in SCRtool with a question mark to represent that the status of these tests are unknown. Then, the checks can be executed by hitting the appropriate button. If the test is successful, it will replace the question mark with a green checkmark; otherwise, it will be replaced with a red X to indicate failure.

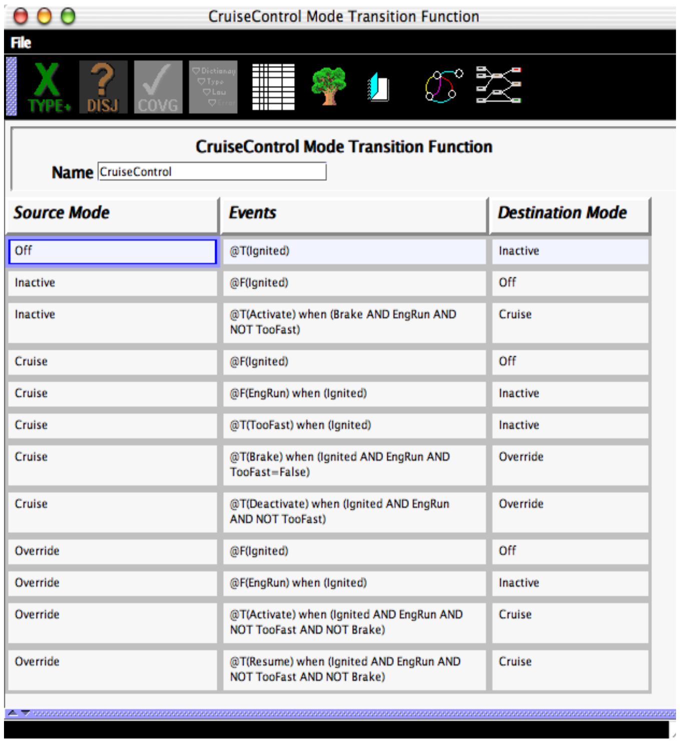
Syntax and Type Checking

The syntax checker will ensure that the specification satisfies the language requirements at a basic level. For example, the syntax checker will ensure that no two variables share the same name or that no illegal characters are used in the variable name. The syntax checking is performed by the Java front-end in real-time. Therefore, as the specification is being edited, the user is being informed as to whether the current specification passes the basic syntax checks or not.

SCRtool may also verify that all instances of variables are used correctly. If a variable is potentially assigned a value that is illegal for its type, an error is emitted. Verification that the correct variable type is used is also present. For example, this means that in a condition function, only term variables are allowed or that a controlled variable is only used in an event function. When variables are used in expressions in a function or another variable, it ensures that only legal comparisons between equivalent types occur.

Disjointness

The disjointness checker verifies that for all term and controlled variables there are no values in common that would lead to an undetermined state. For example, in Table 4, which lists a condition function for a term variable, the values of *Ready* can not overlap. If *Cond1* is true and



CruiseControl Mode Transition Function

Name: CruiseControl

Source Mode	Events	Destination Mode
Off	@T(Ignited)	Inactive
Inactive	@F(Ignited)	Off
Inactive	@T(Activate) when (Brake AND EngRun AND NOT TooFast)	Cruise
Cruise	@F(Ignited)	Off
Cruise	@F(EngRun) when (Ignited)	Inactive
Cruise	@T(TooFast) when (Ignited)	Inactive
Cruise	@T(Brake) when (Ignited AND EngRun AND TooFast=False)	Override
Cruise	@T(Deactivate) when (Ignited AND EngRun AND NOT TooFast)	Override
Override	@F(Ignited)	Off
Override	@F(EngRun) when (Ignited)	Inactive
Override	@T(Activate) when (Ignited AND EngRun AND NOT TooFast AND NOT Brake)	Cruise
Override	@T(Resume) when (Ignited AND EngRun AND NOT TooFast AND NOT Brake)	Cruise

Figure 7: Mode Transition in SCRtool

Cond2 is false, then *Ready* is true. Then, if *Cond1* is false and *Cond2* is true, then *Ready* is false.

This means that there is not an ambiguity about what the value of *Ready* will be based upon the values of *Cond1* and *Cond2*. If an ambiguity were to arise where there was overlap between the TRUE and FALSE state, then this may lead to non-deterministic behavior by the system.

However, if the ANDs were changed to an OR in Table 4, then it would fail the disjointness checks. This would mean an inconsistency would arise if *Cond1* were True and *Cond2* were True. *Ready* would satisfy both of these conditions. Therefore, it is not possible to deterministically decide what the value of *Ready* should be in this situation.

In order to perform the disjointness checker, all of the basic syntax tests must first pass successfully. Otherwise, the disjointness tests can not be executed. The disjointness checks are conducted by the testtool C++ backend, not by the Java front-end. This means that the tests must be explicitly executed by the user in order to obtain the results.

Coverage

The coverage checks are similar to the disjointness in the interface that SCRtool utilizes. The coverage checker is meant to identify any term and controlled variables that do not have a value for a particular state. This could lead to instances when a value is in unknown state. This is slightly different from disjointness in that no value can be derived, but with disjointness, multiple values are legitimate in a particular state.

While it may appear that a condition or event function may not have complete coverage by looking at the table, it is possible to use assumptions about the environment in order to restrict the range of these functions in order to maintain coverage. Therefore, the SCRtool coverage checker must account for the assumptions in order to accurately perform the coverage checks.

In order to understand coverage, we can examine Table 4 again to see if it is covered. In the example, the case with *Cond1* and *Cond2* are both true or both false is not detailed, so the value of *Ready* would be unknown. However, if additional assumptions constrain *Cond1* and *Cond2* to being mutually exclusive, then this function would be covered in this system.

If additional assumptions were not imposed to make the formulas in Table 4 covered, then the events or conditions would have to be altered to ensure that the case when both values are true or both values are false is determined by this function. If they are not covered, then it would be possible for the system to enter an undetermined state.

Prover

SCRtool also includes a prover to check assertions in a specification. Assertions differ from assumptions in that they are desired to be true, but they are not intrinsically true. Assumptions do not need to be proven explicitly, because they represent an inherent facet of the environment that is not mutable. Therefore, to raise the confidence in a specification and its system, the user may prove that the assertions are valid for all legal states in the system.

Each assertion that is desired to be proven must be explicitly enabled. There are multiple states that each assertion is allowed to be in using SCRtool. First, the assertion may be enabled or disabled. If an assertion is disabled, it will be ignored by all components within SCRtool. If it is enabled, it will be verified in external checkers, such as SPIN, and SCRsimulator.

Additionally, there is a Prove enable/disable button. When the assertion is enabled for proving, the prover within SCRtool will attempt to validate the assertion when the prove checks are requested. If the assertion is proved false by the static prover within SCRtool, the sequence of events that fails the assertion will be displayed to the user.

In order to optimize the static prover over a large specification, the user may specify that a statement has already been proved by leaving the prove state disabled. SCRtool will then not seek to prove the variable when the prover is executed, but will assume that it has been previously proven. This will reduce the computation cost of proving and allow for incremental development of the specification since it does not have to recompute costly assertions upon each static prover invocation.

Verification with SPIN

SCRtool allows for the specification to be explored with the SPIN model checker[8]. Currently, SPIN is distributed independently of SCRtool, so the user is required to install it in addition to SCRtool. Once SPIN is installed, SCRtool can translate the current SCR specification as modeled into SPIN's corresponding lanaguage PROMELA. Then, SPIN will examine the specification over a certain range to see if any of the assertions have been violated. If errors are found, an error trail is produced and can be displayed within the SCRtool interface.

SCRtool's integration with SPIN allows for the user to control the parameters to SPIN via the SCRtool interface. The user can modify the maximum search depth, the state space size, and maximum memory usage that the checker will not exceed. Through this interface, the user can also control how SPIN will search the state space - exhaustive, supertrace, or hash-compact. Each of these options is further documented in SPIN's documenation and tutorials[16].

Simulations

Once a specification is created, it is possible to create an executable specification that then can be fed into SCRsimulator. Figure 8 shows a screenshot of SCRsimulator. These simulations are not meant to be exhaustive nor proof that the system is verifiably correct. Instead, SCRsimulator is meant to provide a high-level mockup of the system that potential users can interact with in order to obtain a feel for how the system will behave if it is based on the current state of the specification.

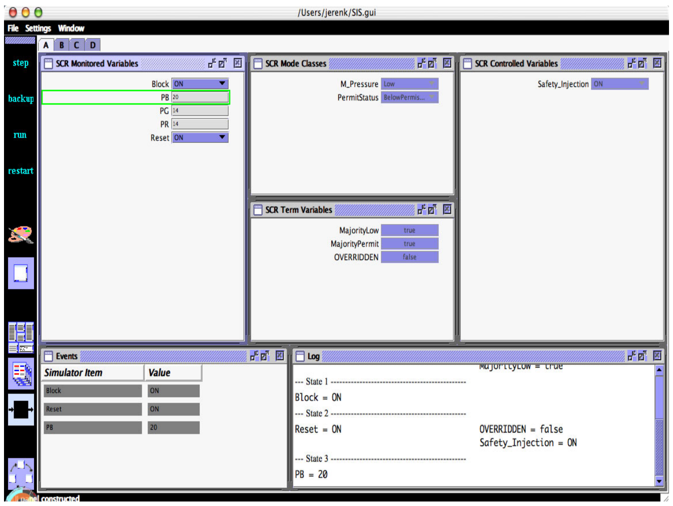
The usage of this simulator allows for rapid prototyping during the specification development process. As the specification is being written, the current model view can be presented to potential users or system experts via SCRsimulator to obtain their feedback as to whether the system is proceeding in the correct direction. However, this is an informal process and does not guarantee that the specification is correct using formal methods.

SCRsimulator allows the users the ability to alter the monitored variables while the system is running. This allows inspection as to how changes in the monitored variables will affect the current mode, as well as the term and controlled variables of the system. SCRsimulator also creates a log of all of the interactions. This logfile allows the events to be later replayed by other users, or for the current user to rewind the simulation to a previous state.

As errors arise during the simulation, SCRsimulator will emit warnings about failed assertions or logical errors that have been violated. This allows the specification creators to correct the specification based upon usage of the system. It may be possible that assumptions could be violated during the simulation. If the system should handle this situation that currently violates the assumption, then the specification can be altered in order to widen the scope of the assumptions accordingly.

3.4. Examples

In [2], a system modeling a cruise control system of an automobile was introduced. Table 5 summarizes the specification using a SCR mode transition table. This system has been partially adopted as a standard challenge problem in the SCR domain. For this example, an independent Java-based implementation of this behavioral specification is



Block	ON
PB	20
PG	14
PR	14
Reset	ON

M_Pressure	Low
PermitStatus	BelowPermis...

Safety_Injection	ON

MajorityLow	true
MajorityPermit	true
OVERRIDDEN	false

Simulator Item	Value
Block	ON
Reset	ON
PB	20

```
                                           MajorityLow = true
--- State 1 ------------------------------------------------
Block = ON
--- State 2 ------------------------------------------------
Reset = ON                                 OVERRIDDEN = false
                                           Safety_Injection = ON
--- State 3 ------------------------------------------------
PB = 20
```

Figure 8: Screenshot of SCRsimulator

freely available on the Internet[3]. This particular implementation is focused on automatically generating tests from SCR-style formal specifications.

Since SCR specifications are necessarily derived from the underlying environment, in order to understand this example, we need to be informed about the nature of an automotive cruise control system. This is a crucial component to any SCR usage - requirements in SCR may only be valid within a certain environmental context.

An automotive cruise control system allows the driver to set a speed that the car will attempt to maintain until it is overriden by the driver. The driver may choose to override the cruise control by either disabling the cruise control explicitly, pressing the gas pedal to accelerate the car temporarily, or engaging the brake pedal and suspending the cruise control until the driver reactivates the cruise control functionality.

In Atlee's example, the cruise control system is modeled by several components in the car: the ignition, an engine on/off button, the speedometer, the gas pedal, the brake, and the cruise control button. This cruise control button has three states: off, cruise, and resume. The cruise control system may also be in four different states: off, inactive, cruise, and override.

4. Alloy Observations

We made the following observations about Alloy as we used the tool. It can definately play a positive role in analyzing system designs. However, we found that there are some issues of caution that must be addressed when using the tool.

4.1. Appropriateness: Who, When, and How

The Alloy langauge is very expressive. However, the cost of its expressiveness is that it also requires a high degree of knowledge to use effectively. Interpreting and writing a correct specification is error prone simply because it requires translation of a cognative concept to and from a formal statement. The required knowledge and the effort in writing and interpreting specification makes the initial cost of specifying a system in the Alloy language high.

The benefits, however, of this investment are potentially two-fold. The first benefit is that the system will likely be less error prone since Alloy will undoubtedly find errors that were initially overlooked. This will translate into fewer maintenance costs later on in a product's lifecycle.

The second cost beneifit is that an abstract model of the system now exists that will be invaluable when evolving the system beyond its initial specification. This will help make modifications more stable since Alloy can check the new modifications against the original specfication for compatibility.

This translates into a high upfront cost for the initial specification of a system. And a lower cost when modifying a system. This suggests that Alloy would be best suited for systems where maintenance costs are significant and where evolvability is important. It also suggests that Alloy is a poor choice for a system with a short lifetime or a where evolvability is not important.

4.2. Scalars, Tuples, and Sets

The fact that Alloy does not differentiate between scalars (an atom), tuples (an ordered set of atoms), and sets of tuples (a relationship) makes the language much simpler. The choice here to treat these three types similarly works because Alloy is not a programming langauge. As a specification language, the differences between these three types is insignificant. Ignoring them allows a specifier to focus on just the core of what they are trying to say without having to worry about all of the syntax involved to make the types match. This also reduces the clutter of the resulting specification, possibly making it easier to comprehend.

4.3. Refutation vs. Proof

Ideally, we would like to prove correctness of every design and program, however, this is not practical. Instead of proofs, Alloy finds counter-examples which refute the correctness of a specification. When finding counter-examples, the state space is limited in scope and the failure to find a counter-example does not mean that there is no error in the system.

On the other hand a counter-example can usually be found very quickly and can be applied towards systems that are not deterministic. The important question to ask is whether a counter-example with its uncertainty is valuable. According to [9], the compromise is worthwhile. Our experience with the tool confirms this. We found that the counter examples provided useful information about flaws in a specification. Additionally, the counter-examples typically had enough information for us to determine what was wrong with the system quickly.

Table 5: Cruise Control ModeClass Transition Table

Current Mode	Ignited	Running	TooFast	Brake	Activate	Deactivate	Resume	New Mode
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	t	t	-	f	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	-	@T	-	
Override	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	-	-	Cruise
	t	t	-	f	-	-	@T	
Initial Mode: Off								

4.4. Graphical Visualization

The graphical visualization of examples and counter-examples is useful at times, but confusing at other times. The graphs can become so cluttered that it is difficult to identify the relationships among the objects. The fact that the look of the graph can be customized helps address this. However, what is really needed is the ability to move graphical components around on the canvas so that they make more sense to the user of the tool.

5. SCR Observations

The following are some observations that we have made by evaluation of the tools and processes that are associated with SCR. SCRtool can definitely help with the automation and management of the requirements process, but there are several concerns that present an obstacle for seeing widepsread adoption in the software engineering community of the SCRtool set and techniques.[4] If these concerns are addressed, then it may be possible to promote the adoption on a wider scale than is currently seen.

5.1. Helps automation

By having tools available to help practicitioners of SCR, it makes it easier to manage the evolution of the system and its associated requirements. Without having automated tools available, managing requirements in a tabular format may become quite cumbersome. As one dependency or expression changes, it requires that all of the dependencies be reexamined. This process does not lend itself well towards a manual process.

In addition to the managing the dependencies with an automated tool, it can also assist in the verification and consistency checking. Manual verification has proven not to be as effective as automated checking of SCR requirements.[6] Therefore, the SCRtool set addresses this concern by offering a lot of automated checkers that have been previously covered in Section 3. Relying upon the automated checking will improve the overall process in comparison to using manual checking.

5.2. Lack of available resources

An issue that faces someone wanting to learn SCR is that there is a dearth of community and public resources around SCR. Currently, it is impossible to obtain the SCR-tool or SCRsimulator code without having contatcs within the SCR community. Therefore, there is not a chance for a public community to be developed around the SCR toolset.

While there are a lot of academic papers available on SCR, they have evidenced the evolution of SCR as it has changed within NRL based on their own experiences using the technique. The SCR described by [7] is indeed subtly different than a contemporary description of SCR as in [6]. Therefore, reading the older papers on SCR does not reflect the current state of the art in using the associated techniques. This presents a serious challenge in understanding what should be done when adopting SCR.

A creation of a public website that describes SCR and acts as a hub of information to the current wealth of knowledge around SCR would be extremely benefical for people who want to learn about SCR. A good website would provide clear examples and well-defined terminology that can aid understanding of the SCR processes.

Additionally, it would also be beneficial if the SCRtool set were distributed publically in an open manner. This would allow people outside the current SCR community to use the toolset, and contribute to the enhancement and fixes of the tools. This would greatly increase the exposure of the tool and the technique, and would hopefully have the additional benefit of allowing the tools to be developed in a collaborative and decentralized fashion where the toolset lives up to its expectations.

5.3. Difference in syntax between tools

Another concern is that there is no universally agreed syntax for SCR. For example, [2] uses a slightly different semantic for some of the mode transition tables than would be utilized by the SCRtool from NRL. This difference makes it difficult to compare instances of SCR requirements created by different tools as each one has a slightly different interpretation of what belongs in the specification.

This issue could be partially resolved by creating a unified format that formally describes the SCR language and techniques. Until now, there have only been partial descriptions of the SCR language and semantics. [6] provides an overview of the techniques currently used in SCR, but does not provide a formal definition. Therefore, the creation of a complete reference guide that explains all of the concepts would be of tremendous benefit to potential adopters.

5.4. Tool interface is shaky

When using SCRtool, the user interface displays some flaws that impairs the usability of the tool. For example, in the version that was evaluated, it is not possible to delete rows from a table. While it is possible to modify the contents of the row or to add a row to the table, a row can not be deleted using the graphical interface. The only solution that has been found at this point is to hand-edit the SCRtool specification file to remove the offending row and reload the specification. While this is only one relatively minor problem, it is indicative of the current quality of the interface.

Additionally, SCRtool requires adhering to their installation configuration in order to work. If the user tries to install SCRtool in a different directory than the one recommended by the NRL, it will require customization by the user that may only be able to be performed by expert users.

These interface and installation difficulties present a challenge to users who are trying to learn SCR via SCRtool. They may experience extreme frustration at learning the tool and the techniques due to the lack of polish in the current tool. For someone who is already familiar with the SCR techniques, they may be willing to look beyond the tool interface deficiencies since they are trying to perform tasks they may already know how to do manually or with another tool. But, for people who are new to SCR, they may be more reliant upon the tool interface to assist them in their requirements and specification gathering. If the tool does not assist them appropriately, they may become frustrated with the SCR technique and avoid it in the future.

5.5. Narrow scope

A fundamental concern with SCR is that it is extremely focused upon a particular activity. A single mode transition table will only represent one selected part of the overall system. This may make it difficult to obtain a coherent perspective of the overall system from just one table. However, this deficiency is balanced by the fact that inspecting one table provides a clear picture of that specific area.

This tradeoff is further complicated by the fact that SCR does not directly support the concept of abstractions. For example, it is not possible using SCRtool to build up a specification that combines mulitple modeclasses. Each modeclass must remain independent from the other modeclass in the specification. If layering and composition of modeclasses were allowed, it might make the task of creating large-scale specifications in SCR a more appealing solution.

6. Comparison between Alloy and SCR

We now compare these tool with respect to what sets them apart form each other. We found ten points of interest when comparing the tool outlined in Table 6. We now discuss each point in detail.

6.1. Objectives

Alloy searches for a model or state that satisfies the constraints of a system. Alloy supports the use of complex data structures using sets and relations. This allows one to describe states directly since the data structures can describe expected data values for the state which are closer to actual values of the desired system. It also allows state transitions to be described declaratively meaning that a state transition is described in more detail than simply a set of assignment values for variables.

SCR is a technique for describing the external behavior of a system and how it reacts to these events. Rather than defining the internal state of the system components, SCR will only capture the externally visible properties of the system. SCR relies upon a paradigm of tables to convey the meaning of the specification instead of using a language construct. This combination of externally visible properties and tabular format allows a separation of concerns in that it does not overspecify the system - a point of concern for many formal methods.[18]

6.2. Lifecycle Stage

While these tools may be used at other stages of the life cycle, we feel that there is a stage where these tools are most effective. Alloy fits somewhere inbetween the design and implementation. SCR works better in the early stages of a software's lifecycle.

Alloy works best after the design stage and before implementation because it can be best used as a formal method to analyze the design. The language Alloy uses to specify a system of constraints is very close to code but could be used at a higher level of abstraction than the implementation code itself. It is possible to use Alloy at earlier stages simply because it is so expressive, however, it seems most appropriate to use as a tool to iteratively analyze a system's design, find counter-examples, then correct the design and Alloy specification accordingly.

SCR works better as a tool to help organize, analyze, and find missing requirements. SCR often represents key concepts in tabular format, which makes it immediately

Table 6: Alloy & SCR Feature Comparison

Feature	Alloy	SCR
Objectives	verifying constraints	behavioral specifications
Life Cycle Stage	after design before implementation	after requirements and before design
Approach	state properties & transitions concrete instances	mode transitions abstract classes
Visibility	defined by modules	derived from tables and cells
Language	very close to a programming language using mathematical expressions	table driven boolean expressions in SCRTool
Reuse	promotes reuse via modules	does not facilitate reuse
Assertions	span multiple states	verify current state
Results	examples satisfying constraints counter-examples violating assertions no proofs	limited counter-examples of violations proof that system will always be valid
Coverage	no coverage	coverage
History	complete system information displayed	relevant system information displayed

apparent which variables are not accounted for in certain mode transitions. This type of organization, in and of itself, is useful as a means of determining what issues may remain overlooked. Because SCR describes *external* behavior of a system, the insights gained into a system using SCR may not translate as readily to the design or implementation of that system since a design deals with both external and internal representations of that system and the implementation is internal.

6.3. Approach

Alloy accomplishes its objectives by analyzing a model describing the properties of each state of a system as well as the properties of each state transition. During analysis, it creates concrete instances of each different types of variables in a system and uses them to construct a finite state space model.

SCR focuses more on the properties of mode transitions and the types of classes involved in the system. SCR determines how each abstract class will affect the mode of the system, and whether the specification covers all possible modes and interactions.

6.4. Visibility

By visibility, we refer to how well one can read, write, correlate understanding of a specification written using one of these approaches. A high degree of visibility allows someone to easily see how different factors of the system interact to produce the results that the tools demonstrate. A low degree of visibility would prevent someone from gaining this valuable insight. Both tools address this issue.

Alloy partially addresses visibility using modules. A module may split a specification into smaller fragments and provides a qualified name that uniquely identifies the elements of that fragment. However, there is no abstraction between modules and all entities in them are considered public in that there is no information hiding. This could easily lead to a specification that has information scattered across multiple modules. While a carefully designed specification could avoid this, Alloy does not enforce this.

SCR suffers from similar visibility problems in that everything is public and the degree to which a specification is truly modular is left to the user to decide. One could conceivably specify an entire system in one large table. It may be more difficult to split a specification into interrelated tables. A benefit of SCR over Alloy is that by looking at a column in a table, it is easier to see how a variable affects different mode transitions. The tabular layout makes the relation between any variable and mode clear and easy to read, which can be very useful when trying to determine how a particular variable affects the system. The fact that all variables affecting a mode are co-located on a table also helps in relating variables to each other. Alloy doesn't have any concept such as this that highlights relationships between different aspects of the system.

6.5. Language

Alloy uses a language that is very close to code. It is therefore very powerful in expressing different concepts and may correlate well to both the design and implementation artifacts. Statements in Alloy consist of boolean algebra, set theory, quantifiers, and first-order relational logic. Consequently, it may be difficult to write or interpret Alloy specifications due to the inherent complexities of these mathematical languages. Certain expressions can be very complex and may need comments to help a user of the specification understand it.

It's clear that Alloy requires someone with a technical background to write and interpret specifications. They must be competent in the areas of mathematics that Alloy uses. While this makes the language more expressible, it limits the number of people that can make use of the tool without prior knowledge. This is a natural trade-off and is probably appropriate for the stage in which Alloy is best used, that is after design and before implementation.

SCR uses a tabular format that allows a user to think in terms of two things at a time. Each cell represents a variable and a mode transition. Since it is generally easier to think about a small set of things at one time than try to juggle a large collection of things, SCR naturally helps a specification to be simpler by taking the tabular approach. In SCR a cell may have one of only five values: t (already true), f (already false), @T (became true), @F (became false), and '-' (currently irrelevant) indicating the relationship of that variable to the mode transition. This makes relating a variable to mode much easier and comprehensible. SCRtool additionally allows a cell to have boolean expressions, enhancing the expressability of the tool. There is also an SCR specific vocabulary that must be learned.

Because SCR is best applied early on in the life cycle, the limitations in expressability are appropriate. A person with little mathematical background could more easily learn how to interpret an SCR table than a complex Alloy expression. It is possible that a customer might fit this description and that they would be interested in viewing an SCR table.

Both tools support the concept of facts about a system (that is statements that are always true) and assertions about a system (that is statements that verify the integrity of the system). Alloy refers to these as 'facts' and 'assertions', while SCR refers to them as 'assumptions' and 'assertions'.

6.6. Reuse

Modules in Alloy provide a limited mechanism for reuse. However, Alloy does not provide a mechanism for hiding information in a module to make it more partitioned. All entities in the module are considered public. This makes it difficult for a module to be tweaked and applied to a different system since other modules may depend on something that was changed.

SCR's reuse is more limited. It is possible that a table that defines the relationship between modes and variables for a very specific component may be reusable. However, reusing a table in a different specification would require that the environments for that table be very similar since SCR's focus is on interation with the environment and the tables are therefore tied to the environment of the sytem.

6.7. Assertions

Both tools provide a means of stating assertions. Alloy, however, allows assertions to span multiple states. An assertion in Alloy can check that a condition would still hold after future state transitions have occurred. This can be seen in the example of a counter-example in section 2.4. In this example, the assertion executes other functions on the current state and determines what the state will be in the future. This is very useful, because it allows assertions to be more expressive

Assertions in SCR are limited to the current (and possibly the previous) mode. Assertions can only be used to make a claim about whether the current mode is valid or not. This may make it difficult to express an assertion that would better be expressed by referring to values of variables as they would be in future or past modes.

6.8. Output

Alloy produces as output examples showing models that satisfy constraints, and counter-examples showing models that violate assertions. As such Alloy can only refute, but not verify that a system is designed properly. This has both advantages and disadvantages. This is advantageous because it allows checking of systems that may not be deterministic. In such a system, it is still useful to determine where the system fails. This can be disadvantageous because it requires a search through a state space. Choosing the right size of the state space is not automatable [9]. While finding no errors in a small state space does not prove the correctness of a program, practice has shown that most errors will be found in a relatively small state space. So, despite the fact that Alloy cannot prove total correctness, it is still a very useful tool.

SCR on the other hand can prove that according to the specification no illegal states will be reached. It does this as part of the process of entering the specification. While entering the specification, SCR analyzes it and highlights problematic issues. SCR does a proof in the sense that if there is an error in the specification, SCR will find it. SCR won't identify a problem if something is left out of the requirements all together. However, it will identify areas where an included variable is inconsistently addressed within the specification.

6.9. Coverage

Alloy does not prove a system to be consistent, it only refutes the claim. When Alloy fails to find a counter-example within a given scope, it cannot be assumed that Alloy will not be able to find a counter-example in a larger scope. In practice this is not really a problem because an error in the system can generally be found in a smaller scope. Additionally, Alloy is applicable to systems that are not provable because they are themselves undecidable. Refuting the correctness of such a system is very useful in those cases.

Since SCR focuses on the transitions between modes, it can cover all transitions completely. Instead of analyzing all of the possible states that can be reached in the specification, SCR will analyze all of the potential state transitions for correctness. This allows specifications that have loops to be analyzed deterministically and in finite time. This is a crucial benefit to ensuring that the specification does satisfy all of the given constraints for all possible cases.

6.10. History

Errors identified by Alloy always contain the complete system information. They show each variable involved in the error. SCR, however, does not typically show this information. It is possible to use the underlying SAT, SPIN, to produce a complete description of how the error was found by viewing its error trails. However, this information is not currently well-integrated into SCRtool and may present challenges to understanding what the tool is exactly providing with these error trails.

7. Discussion

We have examined the functionality offered by Alloy and SCR in order to provide a picture of what these tools can achieve in the arena of formal methods. Furthermore, we have conducted a comparison between SCR and Alloy to understand where each tool belongs in the software development process. We have concluded that SCR is typically better suited to be utilized at an earlier stage of the software lifecycle than Alloy should be.

This disparate role in the software lifecycle may mean that Alloy and SCR can be viewed as complementary tools instead of directly competing tools. Therefore, rather than being forced to choose between the tools, an intelligent user may be able to use the tools together to achieve a clearer picture of how the system will behave.

Certain aspects of Alloy are better suited for some sorts of systems than SCR, and vice versa. Alloy is more expressible than SCR in that a wider variety of concepts can be captured in the specification language. Alloy also offers the ability to provide clear counter-examples that violate the assertions, while SCR focuses on displaying only the local constraint that has been violated.

SCR allows for higher confidence in the fact that the specification can not be violated given that model. Alloy can not definitively prove that a specification has no contradictions or that it might contradict itself. The use of SCR-simulator allows an interactive experience with the specification which broadens the accessibility of the specification in its current format. Alloy, however, is geared more towards programmers in that its constructs require a solid mathematical background.

However, we feel that a more precise evaluation could be conducted by attempting to create a mapping between these languages. If the concepts of one tool can be fully expressed in the other, then we know that the second tool is just as expressive as the first one. If the reverse mapping can also be achieved, i.e. from the second tool to the first tool as well, then that would show to believe that these tools are able to represent equivalent concepts.

Currently, there is no mapping from either Alloy to SCR or SCR to Alloy. However, SCRtool currently includes a mapping from SCR to SPIN's PROMELA. If this mapping could be modified in an automated fashion to generate Alloy-specific constructs, then the first half of our above hypothesis would hold true. We believe that this would likely be possible to achieve based upon the semantic similarity between PROMELA and Alloy.

However, even if the mapping from SCR to Alloy could be achieved, this is not enough to prove that they are equivalent. It is our current belief that there are significant challenges to creating a mapping from Alloy to SCR. These include the challenge of capturing state semantics in SCR and that Alloy may fundamentally be more expressible than SCR. By definitively either proving or disproving that such a mapping can exist will provide us an even better comparison between these tools than presented here.

8. References

[1] Alspaugh, T.A., Faulk, S.R., Britton, K.H., Parker, R.A., Parnas, D.L., and Shore, J.E. *Software Requirements for the A-7E Aircraft*. Naval Research Laboratory, NRL Memorandum Report 3876, August, 1992.

[2] Atlee, J.M. and Buckley, M.A. A Logic-Model Semantics for SCR Software Requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*. p. 280-292, San Diego, California, 1996.

[3] Black, P.E. Cruise Control Example. National Institute of Standards and Technology, 1998. <http://hissa.nist.gov/~black/FTG/CCsim/corvette.html>.

[4] Chechik, M. SC(R)^3 - Towards Usability of Formal Methods. In *Proceedings of the CASCON '98*. November, 1998.

[5] Heitmeyer, C.L., Kirby, J., Labaw, B., and Bharadwaj, R. SCR*: A Toolset for Specifying and Analyzing Software Requirements. In *Proceedings of the Computer-Aided Verification, 10th Annual Conference (CAV'98)*. Vancouver, Canada, 1998. <http://chacs.nrl.navy.mil/publications/CHACS/1998/1998heitmeyer-CAV98.pdf>.

[6] Heitmeyer, C.L. Software Cost Reduction. In *Encyclopedia of Software Engineering*, Marciniak, J.J. ed., 2002.

[7] Heninger, K.L. Specifying Software Requirements for Complex Systems: New Techniques and Their Applications. *IEEE Transactions on Software Engineering*. 6(1), p. 2-13, January, 1980.

[8] Holzmann, G.J. The Model Checker SPIN. *IEEE Transactions on Software Engineering*. 23(5), p. 279-295, May, 1997.

[9] Jackson, D. and Damon, C.A. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*. 22(7), p. 484-495, 1996.

[10] Jackson, D. Automating First-Order Relational Logic. In *Proceedings of the ACM SIGSOFT Conference Foundations of Software Engineering*. p. 130-139, San Diego, California, November, 2000.

[11] Jackson, D., Schechter, I., and Shlyakhter, I. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of*

the 22nd International Conference on Software Engineering. p. 730-733, Limerick, Ireland, June, 2000.

[12] Jackson, D. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 11(2), p. 256-290, April, 2002.

[13] Jackson, D. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy.* <http://sdg.lcs.mit.edu/alloy/reference-manual.pdf>, Software Design Group, PDF, 2002.

[14] NASA Langley Research Center. *What Is Formal Methods?* <http://shemesh.larc.nasa.gov/fm/fm-what.html>, HTML, 2001.

[15] Parnas, D.L. and Madey, J. Functional Documents for Computer Systems. *Science of Computer Programming*. 25(1), p. 41-61, October, 1995.

[16] Ruys, T.C. SPIN Beginners' Tutorial. In *Proceedings of the SPIN 2002 Workshop*. Grenoble, France, April, 2002. <http://spinroot.com/spin/Doc/SpinTutorial.pdf>.

[17] Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice-Hall International Series In Computer Science. 155 pgs., Prentice-Hall International: Englewood Cliffs, N.J., 1989.

[18] Wing, J.M. A Specifier's Introduction to Formal Methods. *IEEE Computer.* 23(9), p. 10-23, September, 1990.