Improving the semantics of the Software Cost Reduction method

Imen Bourguiba a,*, Belaid Moa b

a McMaster University, Hamilton, Ontario, Canada
b University of Victoria, Victoria, B.C., Canada

ABSTRACT

Although the Software Cost Reduction (SCR) method has been successfully used in many projects and organizations to specify software requirements, surprisingly, its semantics is not well defined. The symbols used in this method are ambiguous, especially those that serve to denote SCR events. The aim of this work is to address this ambiguity and improve the SCR semantics by enabling events in first-order logic via two symbols pred and succ. This slight extension of first-order logic allows us to increase the readability of the SCR tables, eliminate their ambiguous semantics, facilitate the verification and validation process, and improve the toolset supporting the SCR method, just to name a few. Moreover, our extension is simple and avoids the complexity of temporal logic.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Table-based specification techniques are both readable and convenient. They allow the representation of system specifications in a very compact and yet precise manner. They scale to software systems, and they may be easily used even by people unfamiliar with the application domain.

In particular, the Software Cost Reduction (SCR) method has been successfully used in practice, especially to formally specify software requirements. It has substantially been improved and extended. Although SCR has been used in many projects and organizations to specify software requirements [3], surprisingly, its semantics is not well defined. The aim of this work is to address this issue. Many advantages are gained by improving the SCR semantics. For instance, the task of verification and validation will be easier. Also, the SCR toolset supporting the method will be improved.

While SCR and our extension can be handled under propositional logic, we choose to extend first-order logic instead as it is much richer. Although temporal logic or event calculus [6] can be used to formalize SRC, our extension does so in a simple and intuitive manner while avoiding the complexity of temporal logic.

The next section is devoted to a general description of the SCR method. In Section 3, the semantics of the SCR method is presented. In Section 4, we depict a new way to model the SCR events with first-order logic. Then a model for the first-order language is given in Section 5. In Section 6, we give an illustrative example to show that tables become readable when our new way to represent SCR events is adopted. Finally, Section 7 draws few conclusions and gives directions for future work.

2. Software Cost Reduction

The Software Cost Reduction (SCR) was originally developed in U.S. Naval Research Lab to document the requirements for the A-7E aircraft [1]. SCR was then successively improved by a team led by C.L. Heitmeyer [3]. It is probably currently the most popular formal method based on a tabular notation for specifying the requirements of software systems. The SCR
requirements specification represents the system behaviour and environment. The environment considers the controlled variables (quantities that the system controls), and the monitored variables (quantities that the system monitors). The environment generates a sequence of monitored events, and the system reacts to these events by changing its state. Under SCR, the system behaviour is represented by a state machine $\Sigma = (S, S_0, E^m, T)$, where $S$ is the set of states, $S_0$ is the set of initial states ($S_0 \subseteq S$), $E^m$ is the set of monitored events, and $T$ is the transform function, which from the current state $s \in S$, and an event $e \in E^m$ specifies the next state $s' \in S$. The SCR state machine model is a special case of Parnas' Four Variable Model (FVM) [7]. There are a couple of slightly different versions of the FVM. The one used here is developed by Parnas and Madey to specify system requirements [8]. It is an extension of the classical “black box” Two Variable Model (input and output) [7]. The SCR formal model uses only the relations $\text{NAT}$ and $\text{REQ}$ to define the system behaviour. The $\text{NAT}$ relation depicts the constraints put on the controlled and monitored variables while $\text{REQ}$ specifies the relations between the monitored and controlled variables.

In order to have a more concise specification, some constructs such as mode classes and terms were added to the SCR model. The values of mode classes are modes; they are classes of system states specifying the system behaviour. With SCR, each specification is organized into dictionaries and tables. The dictionaries represent static information such as variables names and types, whereas the tables depict the changes of the variables with respect to input events. In SCR, there are three kinds of tables to specify a system: condition tables, event tables, and mode transition tables. The tables discussed here describe a safety injections system, and are borrowed from [5]. A condition table defines a variable according to a mode and a condition. A condition is a predicate defined on a system state. For example, Table 1 identifies the controlled variable SafetyInjection as a function of Pressure and the term Overridden. For instance, the first column of Table 1 indicates that if the Pressure is High or Permitted, or if the Pressure is TooLow and Overridden is True, then SafetyInjection is Off.

An event table defines a variable according to a mode and an event. An event represented by $@T(c)$ means that condition $c$ changes from false to true. For example, $@T(\text{Block}==\text{On})$ when $(\text{Reset}==\text{Off})$ means that the operator turns Block from Off to On when the Reset is Off. The $@T(\text{Inmode})$ means that the system enters into the class of modes in that row. In Table 2, the cell $(2, 1)$ indicates that if the Pressure is TooLow or Permitted, and Block changes to On When Reset is Off, then Overridden changes to True.

A mode transition table generates a destination mode from a mode and an event. In the first row of Table 3, we see that if the Pressure is TooLow and WaterPress is greater than or equal to Low, then Pressure becomes Permitted.

### Table 1
Condition table.

<table>
<thead>
<tr>
<th>Mode Class</th>
<th>Pressure Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>High, Permitted</td>
<td>True</td>
</tr>
<tr>
<td>TooLow</td>
<td>Overridden</td>
</tr>
<tr>
<td>SafetyInjection</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>NOT Overridden</td>
</tr>
</tbody>
</table>

### Table 2
Event table for Overridden.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>False @T(Inmode)</td>
</tr>
<tr>
<td>TooLow, Permitted</td>
<td>@T(Block==On) WHEN Reset==Off</td>
</tr>
<tr>
<td>Overridden</td>
<td>True @T(Inmode) OR @T(Reset==On) False</td>
</tr>
</tbody>
</table>

### Table 3
Mode transition table for Pressure.

<table>
<thead>
<tr>
<th>Old Mode</th>
<th>Event</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TooLow</td>
<td>@T(WaterPress $\geq$ Low)</td>
<td>Permitted</td>
</tr>
<tr>
<td>Permitted</td>
<td>@T(WaterPress $\geq$ Permit)</td>
<td>High</td>
</tr>
<tr>
<td>Permitted</td>
<td>@T(WaterPress $&lt;$ Low)</td>
<td>TooLow</td>
</tr>
<tr>
<td>High</td>
<td>@T(WaterPress $&lt;$ Permit)</td>
<td>Permitted</td>
</tr>
</tbody>
</table>

3. SCR semantics

In SCR, a Simple Condition is true, false, or a logical statement of the form $r \odot v$, where $r$ belongs to the set of entity names, $\odot$ is a relational operator, and $v$ is a constant value that belongs to the type of $r$. A Condition is a logical statement composed of Simple Conditions connected by logical connectors. Let us consider an example of a condition borrowed from [4].

\[ r \in \{=, \neq, <, >, \leq, \geq \} \]
The set of entity names RF is defined by:

\[ RF = \{ \text{Block, Reset, WaterPress, Pressure, SafetyInjection, Overridden} \}. \]

The type definitions are:

- \( TY(\text{Pressure}) = \{ \text{TooLow, Permitted, High} \} \)
- \( TY(\text{WaterPress}) = \{ 0, 1, 2, \ldots, 2000 \} \)
- \( TY(\text{Overridden}) = \{ \text{true, false} \} \)

For example, Overridden = true is a simple condition, where Overridden belongs to set of entity names RF, “=” is a relational operator, and the constant \( v \) is true, and it belongs to the boolean type of Overridden.

A Basic Event is denoted by \( @T(c) \), where \( c \) is a simple condition, which state changes from false to true. Similarly, \( @F(c) \) means that condition \( c \) state changes from true to false.

A Simple Conditioned Event is denoted by \( \text{@T(c) WHEN } d \), where \( d \) is a condition.

\[ \text{@T(c) WHEN } d = c' \land \neg c \land d, \]

where the non-primed condition denoted by \( c \) represents its old value, and the primed condition denoted by \( c' \) depicts its new value.

For example,

\[ \text{@T(Block=On) WHEN Reset=Off} \]

can be rewritten as:

\[ \text{Block'} = \text{On} \land \text{Block} = \text{Off} \land \text{Reset} = \text{Off} \]

Finally, a Conditioned Event is composed of Simple Conditioned Events connected by the logical connectors. In what follows, we present our event modelling in First-Order Logic, and the model supporting it.

### 4. Event modelling in First-Order Logic

Let \( L = (C, F, P) \) be a first-order language such that \( C \) is a set of symbols called constants, \( F \) is a set of symbols called function symbols, each with arity \( \geq 1 \), and \( P \) is a set of symbols called predicate symbols, each with arity \( \geq 1 \). \( P \) contains the binary predicate symbol “=”.

Let \( V \) be the set of symbols called variables. \( V, C, F, \) and \( P \) are pair-wise disjoint.

A term in \( L \) is defined recursively as follows: a constant or a variable is a term. Given a function symbol \( f \) with \( n \)-arity and the terms \( t_1, \ldots, t_n, f(t_1, \ldots, t_n) \) is a term.

A formula is defined as follows. Given a predicate \( P \) with \( n \)-arity, \( t_1, \ldots, t_n \) terms, \( P(t_1, \ldots, t_n) \) is a formula. For any formula \( F \), \( \neg F \), and \( \forall x F \), \( \exists x F \) are formulas. Given two formulas \( F_1 \) and \( F_2 \), \( F_1 \land F_2 \) and \( F_1 \lor F_2 \) are also formulas.

Given a language \( L = (C, F, P) \) defined as above, we extend it with two new symbols \( \text{pred} \) and \( \text{succ} \) called event symbols, and we call the new language, \( L_{ev} = (C, F, \text{pred}, \text{succ}) \), an event-enabled first-order logic language.

Intuitively, the new symbol \( \text{pred} \) represents the past (previous) value of a term or a formula, and the symbol \( \text{succ} \) specifies the future (next) value of a given term or a formula. With our event-enabled language, we can easily model an SCR system in which the environment generates a sequence of events, and the system reacts to these events by changing its state.

The new symbols \( \text{pred} \) and \( \text{succ} \) allows us to introduce new terms, formulas and axioms in addition to those of first-order logic.

**Definition 1.** If \( t \) is a term, then \( \text{pred}(t) \) and \( \text{succ}(t) \) are also terms. Moreover, if \( t \) is a constant, we have \( \text{pred}(t) = t \) and \( \text{succ}(t) = t \).

Intuitively, for every term \( t \), \( \text{pred}(t) \) and \( \text{succ}(t) \) depict the past and the future values of \( t \), respectively.

**Definition 2.** For a term \( t \) of the form \( f_n(t_1, \ldots, t_n) \), we have

\[ \text{pred}(f_n(t_1, \ldots, t_n)) = f_n(\text{pred}(t_1), \ldots, \text{pred}(t_n)) \]

and

\[ \text{succ}(f_n(t_1, \ldots, t_n)) = f_n(\text{succ}(t_1), \ldots, \text{succ}(t_n)). \]
\textbf{Definition 3}. If $F$ is a formula of the form $P_m(t_1, \ldots, t_m)$, then $\text{pred}(F)$ and $\text{succ}(F)$ are formulas and
\[ \text{pred}(P_m(t_1, \ldots, t_m)) = P_m(\text{pred}(t_1), \ldots, \text{pred}(t_m)), \]
and
\[ \text{succ}(P_m(t_1, \ldots, t_m)) = P_m(\text{succ}(t_1), \ldots, \text{succ}(t_m)). \]
In particular, for $t_1 = t_2$, we have:
\[ \text{pred}(t_1) = \text{pred}(t_2) \quad \text{and} \quad \text{succ}(t_1) = \text{succ}(t_2). \]

It easy to see that if $\gamma(t_1, \ldots, t_n)$, where $\gamma$ is either a function or a predicate symbol, involves constants only, then $\text{pred}(\gamma(t_1, \ldots, t_n)) = \gamma(t_1, \ldots, t_n)$ and $\text{succ}(\gamma(t_1, \ldots, t_n)) = \gamma(t_1, \ldots, t_n)$.

\textbf{Definition 4}. Given two formulas $F_1$ and $F_2$, we have:
\[
\begin{align*}
\text{pred}(\neg F_1) &= \neg \text{pred}(F_1) \\
\text{pred}(F_1 \lor F_2) &= \text{pred}(F_1) \lor \text{pred}(F_2) \\
\text{pred}(F_1 \land F_2) &= \text{pred}(F_1) \land \text{pred}(F_2)
\end{align*}
\]
and,
\[
\begin{align*}
\text{succ}(\neg F_1) &= \neg \text{succ}(F_1) \\
\text{succ}(F_1 \lor F_2) &= \text{succ}(F_1) \lor \text{succ}(F_2) \\
\text{succ}(F_1 \land F_2) &= \text{succ}(F_1) \land \text{succ}(F_2).
\end{align*}
\]

\textbf{Definition 5}. Given a formula $F$ and a variable $x$, we have:
\[
\begin{align*}
\text{pred}(\forall x F) &= \forall x \text{pred}(F) \\
\text{pred}(\exists x F) &= \exists x \text{pred}(F)
\end{align*}
\]
and,
\[
\begin{align*}
\text{succ}(\forall x F) &= \forall x \text{succ}(F) \\
\text{succ}(\exists x F) &= \exists x \text{succ}(F).
\end{align*}
\]

Moreover, $\text{pred}(x) = x$ and $\text{succ}(x) = x$ for the bound variable $x$ in $F$.

Because of Definitions 2, 3, 4 and 5, the symbols $\text{succ}$ and $\text{pred}$ are qualified as being linear.

\textbf{Definition 6}. For every natural number $n$ and every term or formula $\zeta$, we define the $\text{pred}^n$ symbol as follows:
\[
\begin{align*}
\text{If } n &= 0, \quad \text{then } \text{pred}^0(\zeta) = \zeta \\
\text{If } n &\geq 1, \quad \text{then } \text{pred}^n(\zeta) = \text{pred}(\text{pred}^{n-1}(\zeta)).
\end{align*}
\]

Similarly, for every natural number $n$, we define the $\text{succ}^n$ symbol as follows:
\[
\begin{align*}
\text{If } n &= 0, \quad \text{then } \text{succ}^0(\zeta) = \zeta \\
\text{If } n &\geq 1, \quad \text{then } \text{succ}^n(\zeta) = \text{succ}(\text{succ}^{n-1}(\zeta)).
\end{align*}
\]

Moreover, we take $\text{pred}^{-n}$ to stand for $\text{succ}^n$ and $\text{succ}^{-n}$ for $\text{pred}^n$.

In the above definition, if we take $n$ to be 1, then $\text{pred}^{-1} = \text{succ}$ and $\text{succ}^{-1} = \text{pred}$. Therefore, we could have extended FOL with a single symbol, say $\text{pred}$, and use $\text{pred}^{-1}$ instead of $\text{succ}$. We prefer, however, to have both symbols as they make the syntax clearer.

Using this definition, it is easy to see that for every natural number $n$ and every term or formula $\zeta$, we have
\[
\text{pred}^{-n-1}(\zeta) = \text{pred}^{-1}(\text{pred}^{-n}(\zeta)) \quad \text{and} \quad \text{succ}^{-n-1}(\zeta) = \text{succ}^{-1}(\text{succ}^{-n}(\zeta)).
\]
Definition 7. An event on a condition \( c \) is given by
\[
\text{succ}(c) \land \neg c,
\]
or equivalently
\[
c \land \neg \text{pred}(c).
\]

Definition 8. For every variable \( v \), we axiomatize that
\[
\text{pred}(\text{succ}(v)) = v \quad \text{and} \quad \text{succ}(\text{pred}(v)) = v.
\]

Theorem 4.1. For every term or formula \( \zeta \), we have
\[
\text{pred}(\text{succ}(\zeta)) = \zeta \quad \text{and} \quad \text{succ}(\text{pred}(\zeta)) = \zeta.
\]

Proof. The theorem follows from the recursive nature of terms and formulas and the linear property of \( \text{pred} \) and \( \text{succ} \).

If \( \zeta \) is a constant or a variable, then from Definitions 1 and 8 we deduce that
\[
\text{pred}(\text{succ}(\zeta)) = \zeta \quad \text{and} \quad \text{succ}(\text{pred}(\zeta)) = \zeta.
\]

If \( \zeta \) is a term of the form \( f_n(t_1, \ldots, t_n) \), then
\[
\text{pred}(\text{succ}(f_n(t_1, \ldots, t_n))) = f_n(\text{pred}(\text{succ}(t_1), \ldots, \text{pred}(\text{succ}(t_n))))
\]
(from Definition 2). By recursively doing the same for the terms \( t_1, \ldots, t_n \) until they are constants or variables, we obtain
\[
\text{pred}(\text{succ}(f_n(t_1, \ldots, t_n))) = f_n(t_1, \ldots, t_n).
\]

We leave the case where \( \zeta \) is a formula for the reader. \( \square \)

With our definitions, we avoided the prefixed SCR notations with “@T(c)” to represent events by \( @T(c) \) or \( @F(c) \). In fact, it becomes straightforward to express \( @F(c) \) in terms of \( \text{pred} \) and \( \text{succ} \) by the simple formula

\[
\neg c \land \text{pred}(c)
\]
or, equivalently,
\[
\neg \text{succ}(c) \land c
\]

Definition 9. An \( n \)-past event on a variable \( x \) denoted by \( \text{event}^{-n}(x) \) is defined by
\[
\neg (\text{pred}^{-n}(x) = x)
\]
An \( n \)-future event on a variable \( x \) denoted by \( \text{event}^{+n}(x) \) is defined by
\[
\neg (\text{succ}^{n}(x) = x)
\]

An \( n \)-past (respectively \( n \)-future) event indicates whether an event happens between the current and the \( n \)-past (respectively \( n \)-future) time steps.

Definition 10. Let \( f \) be a function symbol with \( k \)-arity, and \( k \) terms \( t_1, \ldots, t_k \). We call an \( n \)-past event of the function \( f \), the formula denoted by \( \text{event}^{-n}_n(f)(t_1, \ldots, t_k) \), and defined by:
\[
\neg (\text{pred}^{-n}(f(t_1, \ldots, t_k)) = f(t_1, \ldots, t_k)).
\]
An \( n \)-future event on the function \( f \) is the formula denoted by \( \text{event}^{+n}_n(f)(t_1, \ldots, t_k) \), and defined by:
\[
\neg (\text{succ}^{n}(f(t_1, \ldots, t_k)) = f(t_1, \ldots, t_k)).
\]

As an example of 1-past event, let “inc” be an increment function which adds the value one to any input value. \( \text{event}^{-1}_1(\text{inc}(x)) \) is defined by:
\[
\neg (\text{pred}(\text{inc}(x)) = \text{inc}(x))
\]
\[
\iff \neg (\text{pred}(x + 1) = x + 1)
\]
\[
\iff \neg (\text{pred}(x) + 1 = x + 1)
\]
\[
\iff \neg ((x - 1) + 1 = x + 1)
\]
\[
\iff \neg (x = x + 1).
\]
This indicates that there is a 1-past event since the value of the variable \( x \) changed.
Definition 11. Given a predicate symbol $Q$ with $q$-arity, and $q$ terms $t_1, \ldots, t_q$, an $n$-past event on predicate $Q$ is the formula $\text{event}_n^{-}(Q)(t_1, \ldots, t_q)$ defined by:
$$\neg(\text{pred}_n^a( Q(t_1, \ldots, t_k))) \land Q(t_1, \ldots, t_k).$$

An $n$-future event on predicate $Q$ is the formula $\text{event}_n^{+}(Q)(t_1, \ldots, t_k)$ given by:
$$\neg(\text{succ}_n^a( Q(t_1, \ldots, t_k))) \land Q(t_1, \ldots, t_k).$$

From the above definitions, we have the following lemma.

Theorem 4.2. Given a function or a predicate symbol $\gamma$ with $k$-arity and $k$ terms $t_1, \ldots, t_k$, we have:
$$\gamma(S^n(t_1), \ldots, S^n(t_k)) = S^n(\gamma(t_1, \ldots, t_k)),$$
where $S$ is the pred or succ symbol.

Proof. By induction:
1) By using Definition 6, we have
$$\gamma(S^0(t_1), \ldots, S^0(t_k)) = \gamma(t_1, \ldots, t_k) = S^0(\gamma(t_1, \ldots, t_k)).$$
Therefore, the theorem holds for the base case $n = 0$.

2) Let us assume that for some $n$, we have:
$$\gamma(S^n(t_1), \ldots, S^n(t_k)) = S^n(\gamma(t_1, \ldots, t_k)).$$

3) Let us prove that:
$$\gamma(S^{n+1}(t_1), \ldots, S^{n+1}(t_k)) = S^{n+1}(\gamma(t_1, \ldots, t_k)).$$
$$\gamma(S^{n+1}(t_1), \ldots, S^{n+1}(t_k))$$
$$= (\text{By Definition 6})$$
$$\gamma(S(S^n(t_1)), \ldots, S(S^n(t_k)))$$
$$= (\text{By Definition 2 or 3 depending on } \gamma)$$
$$S(\gamma(S^n(t_1), \ldots, S^n(t_k)))$$
$$= (\text{By induction hypothesis})$$
$$S(S^n(\gamma(t_1, \ldots, t_k)))$$
$$= (\text{By Definition 6})$$
$$S^{n+1}(\gamma(t_1, \ldots, t_k)). \quad \square$$

Definition 12. Given a formula $F$, an $F$-guarded $n$-past event on variable $x$ denoted by $x|F$ is defined by:
$$\text{event}_n^{-}(x) \land F$$
An $F$-guarded $n$-past event on function symbol $f$ denoted by $f|F$ is defined by:
$$\text{event}_n^{-}(f) \land F$$
An $F$-guarded $n$-past event on predicate symbol $P$ denoted by $P|F$ is defined by:
$$\text{event}_n^{-}(P) \land F$$

Definition 13. Given a formula $F$, an $F$-guarded $n$-future event on variable $x$ denoted by $x\parallel F$ is defined by:
$$\text{event}_n^{+}(x) \land F$$
An $F$-guarded $n$-future event on function symbol $f$ denoted by $f\parallel F$ is defined by:
$$\text{event}_n^{+}(f) \land F$$
An $F$-guarded $n$-future event on predicate symbol $P$ denoted by $P\parallel F$ is defined by:
$$\text{event}_n^{+}(P) \land F$$
5. A model for the event-enabled First-Order Logic

A model $M$ for the event-enabled first-order language $L_{ev}$ is a two tuple $(D, I)$, where $D$ is a nonempty domain of individuals, and $I$ is an interpretation function that assigns each constant, function symbol, and predicate symbol in $L_{ev}$ over $D$. More details about the model can be found at [2].

Let $\text{Var}$ be the set of variables, $\text{Term}$ be the set of terms of $L_{ev}$, $\text{Form}$ be the set of formulas of $L_{ev}$, and $\text{Int}$ be the set of integers.

To represent past and future values of terms and formulas, the usual definition of assignment or valuation needs to be adjusted. This we do next.

A variable assignment for $M$ is a total function

$$\phi : \text{Var} \times \text{Int} \rightarrow D$$

that maps each variable at some point in time to an element of $D$.

Let $\text{VarAssign}(M)$ be the set of all variable assignments for $M$.

The valuation function for $M$ is the function

$$V : (\text{Term} \cup \text{Form}) \times \text{VarAssign} \times \text{Int} \rightarrow D \cup \{\text{true}, \text{false}\},$$
defined by the following statements, where $\phi \in \text{VarAssign}$ and $i \in \text{Int}$:

- If $x$ is a variable,
  $$V(x, \phi, i) = \phi(x, i)$$

- If $c$ is an individual constant,
  $$V(c, \phi, i) = I(c)$$

- If $f$ is an $n$-ary function symbol and $t_1, \ldots, t_n$ are terms of $L_{ev}$,
  $$V(f(t_1, \ldots, t_n), \phi, i) = I(f)(V(t_1, \phi, i), \ldots, V(t_n, \phi, i))$$

- If $P$ is an $n$-ary predicate and $t_1, \ldots, t_n$ are terms of $L_{ev}$,
  $$V(P(t_1, \ldots, t_n), \phi, i) = I(P)(V(t_1, \phi, i), \ldots, V(t_n, \phi, i))$$

- If $\zeta$ is a term or a formula of $L_{ev}$,
  $$V(\text{pred}(\zeta), \phi, i) = V(\zeta, \phi, i - 1)$$

- If $t_1$ and $t_2$ are terms of $L_{ev}$,
  $$V(t_1 = t_2, \phi, i) = \text{true} \quad \text{if} \quad V(t_1, \phi, i) = V(t_2, \phi, i)$$
  $$V(t_1 = t_2, \phi, i) = \text{false} \quad \text{otherwise}.$$

- If $F$ is a formula of the form $P(t_1, \ldots, t_n)$, where $P$ is an $n$-ary predicate and $t_1, \ldots, t_n$ are terms of $L_{ev}$,
  $$V(\neg F, \phi, i) = \text{true} \quad \text{if} \quad V(F, \phi, i) = \text{false}$$
  $$V(\neg F, \phi, i) = \text{false} \quad \text{otherwise}$$

- If $F_1$ is a formula of the form $P(t_1, \ldots, t_n)$, where $P$ is an $n$-ary predicate and $t_1, \ldots, t_n$ are terms of $L_{ev}$, $F_2$ is a formula of the form $P(t_1, \ldots, t_m)$, where $P$ is an $m$-ary predicate and $t_1, \ldots, t_m$ are terms of $L_{ev}$,
  $$V(F_1 \lor F_2, \phi, i) = \text{false} \quad \text{if} \quad V(F_1, \phi, i) = \text{false} \land V(F_2, \phi, i) = \text{false}$$
  $$\quad \text{otherwise} \quad V(F_1 \lor F_2, \phi, i) = \text{true}$$

- If $F_1$ is a formula of the form $P(t_1, \ldots, t_n)$, where $P$ is an $n$-ary predicate and $t_1, \ldots, t_n$ are terms of $L_{ev}$, $F_2$ is a formula of the form $P(t_1, \ldots, t_m)$, where $P$ is an $m$-ary predicate and $t_1, \ldots, t_m$ are terms of $L_{ev}$,
  $$V(F_1 \land F_2, \phi, i) = \text{true} \quad \text{if} \quad V(F_1, \phi, i) = \text{true} \land V(F_2, \phi, i) = \text{true}$$
  $$\quad \text{otherwise} \quad V(F_1 \land F_2, \phi, i) = \text{false}$$
Table 4
Event table for Overridden with the “pred” symbol.

<table>
<thead>
<tr>
<th>Events</th>
<th>(¬pred(Pressure = High)) ∧ (Pressure = High)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>(¬pred(Pressure = TooLow)) ∧ (Pressure = TooLow)</td>
</tr>
<tr>
<td>∨</td>
<td>(¬pred(Pressure = Permitted)) ∧ (Pressure = Permitted)</td>
</tr>
<tr>
<td>∧</td>
<td>(¬pred(Reset=On)) ∧ (Reset=On) ∧ (¬pred(Reset=Off)) ∧ (Reset=Off) ∧ (¬pred(Overridden)) ∧ (Overridden)</td>
</tr>
</tbody>
</table>

- If $F$ is a formula of the form $\forall x(P(t_1, \ldots, t_n))$, where $P$ is an $n$-ary predicate and $t_1, \ldots, t_n$ are terms of $L_{ev}$,
  
  $$V(F, \phi, i) = true \quad \text{if} \quad V(F, \phi[x \mapsto y], i) = true \quad \forall y \in D$$

  otherwise $V(F, \phi, i) = false$

- A formula $A$ is valid in $M$ if, for $\forall \phi \in \text{VarAssign}$ and $i \in \text{Int}$,
  
  $$V(A, \phi, i) = true$$

**Theorem 5.1.** For every term or formula $\zeta$ in $L_{ev}$, we have

$$V(\text{suc}(\zeta), \phi, i) = V(\zeta, \phi, i + 1).$$

Moreover, for every integer $z$, we have

$$V(S^2(\zeta), \phi, i) = V(\zeta, \phi, i + z),$$

where $S$ is pred or succ.

**Proof.**

$$V(t, \phi, i + 1)$$

$$= \langle \text{By Theorem 4.1} \rangle$$

$$V(\text{pred(suc}(t)), \phi, i + 1)$$

$$= \langle \text{By definition of the valuation function of pred} \rangle$$

$$V(\text{suc}(t), \phi, i).$$

Using induction and Definition 6, it is easy to prove that $V(S^2(\zeta), \phi, i) = V(\zeta, \phi, i + z)$. □

6. Illustrative example

In SCR, some constructs were added to the general SCR model trying to make the specification more concise. So, the developers should be knowledgeable of those concepts, and this makes the task of building the specification difficult. We support our claim by a study on the application of SCR on a Space Station Biological Research Project at the NASA Ames Research Center (ARC) [9]. As a matter of fact, building the SCR specification took a lot of time, and the SCR project team had to intervene, and use their expertise about mathematical and state machine models to build the initial SCR specification, and provide it to the ARC developers for modification and extension.

In this section, we provide an example where we use our semantics, and apply our new definitions for SCR events on an SCR event table. In this example, we transform Table 2 into Table 4, and Table 5 using the pred and succ symbols respectively. In Table 4, we represented the events on a condition $c$ by $c \land \neg\text{pred}(c)$, equivalently, in Table 5, we represented the events on a condition $c$ by $\text{succ}(c) \land \neg\neg$ (Definition 7). Therefore, Table 4 and Table 5 are semantically equivalent.

In Table 2, there are some notations such as @T(Inmode) and @T(Block=On) when Reset=Off that are not easily understood by users/developers who are not quiet familiar with the SCR method. Also, it is not obvious that the term overridden is obtained by the conjunction of the respective mode and event. In our transformed tables, we removed the Mode column, and we write explicitly the mode with its respective event.

From this example, it is clear that Table 4 and Table 5 are easier to interpret than Table 2 which contains constructs that are not in FOL. Hence, with the well-defined semantics that we proposed, our translated tables are now amenable for formal verification and logical reasoning.

In the next section we conclude and give directions for future work.
Table 5
Event table for Overridden with the “succ” symbol.

<table>
<thead>
<tr>
<th>Events</th>
<th>Succ(Pressure = High) ∧ ¬(Pressure = High)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False (succ(Pressure = TooLow)) ∧ ¬(Pressure = TooLow)</td>
<td>(succ(Pressure = TooLow)) ∧ ¬(Pressure = TooLow)</td>
</tr>
<tr>
<td>∨</td>
<td>∨</td>
</tr>
<tr>
<td>(succ(Pressure = Permitted)) ∧ ¬(Pressure = Permitted)</td>
<td>(succ(Pressure = Permitted)) ∧ ¬(Pressure = Permitted)</td>
</tr>
<tr>
<td>∧</td>
<td>∧</td>
</tr>
<tr>
<td>(succ(Block=On)) ∧ ¬(Block=On)</td>
<td>(succ(Reset=On)) ∧ ¬(Reset=On)</td>
</tr>
<tr>
<td>∧ (succ(Reset=Off)) ∧ ¬(Reset=Off)</td>
<td></td>
</tr>
<tr>
<td>(succ(Overridden)) ∧ ¬(Overridden)</td>
<td>(succ(Overridden)) ∧ ¬(Overridden)</td>
</tr>
</tbody>
</table>

7. Conclusion

Although SCR is popular and has been used in many industrial and academic organizations, it has some limitations regarding its semantics, and its symbols are ambiguous [3]. In this work, we addressed this ambiguity, and improved SCR semantics by modeling SCR events in first-order logic. Many advantages are obtained with the conversion that we proposed. The tables are more readable, and could be easily interpreted even by people who are unfamiliar with the domain. Besides, we avoid previous ambiguous symbols (e.g. primed notations and prefixed notations with “@” symbols). Hence, by improving SCR semantics, there are many tasks that could be carried out such as facilitating the verification and validation process, and improving the toolset supporting the SCR method. Our extension can easily be specialized to the propositional logic.

As a future work, for SCR events, we suggest that the conditioned events will be executed by guarded commands where the condition and action will be representing events.

Acknowledgements

We would like to thank the reviewers for their vital comments and suggestions. We also thank Prof. W. Farmer for suggesting the event-enabled First-Order Logic model, and Prof. Maarten van Emde for introducing us to event calculus.

References