The feature construct for SMV: semantics

Malte Plath and Mark Ryan
School of Computer Science
University of Birmingham
Birmingham B15 2TT
UK
`http://www.cs.bham.ac.uk/{~mcp,~mdr}`

1 Introduction

The concept of *feature* has emerged in telephone systems analysis as a way of describing optional services to which telephone users may subscribe [1, 3, 5]. Features offered by telephone companies include call-forwarding, automatic-call-back, and voice-mail. Features are not restricted to telephone systems, however. Any part or aspect of a specification which the user perceives as having a self-contained functional role is a feature. For example, a printer may exhibit such features as: ability to understand PostScript; Ethernet card; ability to print double-sided; having a serial interface; and others. The ability to think in terms of features is important to the user, who often understands a complex system as a basic system plus a number of features. It is also an increasingly common way of designing products.

To support this way of building a system from a basic system by successively adding features, we have extended the syntax of SMV[1] with a feature construct that allows features to be described in a compact way, and we have developed the tool SFI ('SMV feature integrator') that compiles the extended SMV code into simple SMV code which the model checker can work with. We handle the potential inconsistency between a feature and the base system by allowing features to override existing behaviour in a tightly controlled way. We have used SMV to verify a *lift system* (elevator system in the US) together with five features and their interactions [9]. We have also verified a model of the telephone system together with about seven features [8], and the feature interactions between them.

The results in those papers were entirely experimental. Our goal in this paper is to give precise semantics to the feature construct for SMV. Using such semantics, we can explore in what circumstances the feature construct is independent of the syntax of the base program to which the feature is applied, and other properties of features. This allows us to explore when features commute with each other, and, more generally, to explore what classes of features are 'interaction-friendly' with respect to other classes. It is also a step towards being able to verify the feature itself, independently of the system to which it is added.

[1]SMV ('Symbolic Model Verifier') is a model checker developed by Ken McMillan at Carnegie Melon University [7]. It may be obtained from `www.cs.cmu.edu/~modelcheck`. Until 1998 there was just one SMV, but now there are three. CMU SMV is the original one and is the subject of this paper. NuSMV is a re-implementation being developed in Trento [2], and is aimed at being customizable and extensible. Cadence SMV is an entirely new model checker focussed on compositional systems. It is also developed by Ken McMillan, and its description language resembles but much extends the original SMV [6].

The paper is structured as follows. The remainder of this section describes some features from a motivational point of view. Section 2 recalls the feature construct for SMV, first presented in [8]. In order to develop its semantics as a transition system transformer, we first define the semantics of an SMV program as a transition system (section 3). Section 4 then develops the semantics of the feature construct. The conclusions are in section 5.

1.1 Experimental results using the feature construct

One of our case studies with the feature construct is a simple version of the Plain Old Telephone System (POTS). We briefly recall the features we have modelled and integrated (under various combinations) into our model of POTS:

- Call Waiting (CW)
- Call Forward Unconditional (CFU)
- Call Forward on Busy (CFB)
- Call Forward on No Reply (CFNR)
- Ring Back When Free (RBWF)
- Terminating Call Screening (TCS)
- Originating Call Screening (OCS)

A feature comprises two components: the feature implementation δ (described in terms of the keywords 'treat' and 'impose', detailed the next section), and the feature requirements as a CTL formula ϕ. When we integrate a feature (δ, ϕ) into a base system P, we want to test the following:

- $P + \delta \models \phi$: Feature δ has been successfully integrated.
- $(P + \delta_1) + \delta_2 \models \phi_2$: Feature δ_2 can be integrated into the extended system $P + \delta_1$.
- $(P + \delta_1) + \delta_2 \models \phi_1$: Feature δ_2 does not violate the requirements of δ_1.

Of course these tests will not necessarily succeed. We shall however assume that all features are correct wrt. the base system, i.e., $P + \delta \models \phi$ for any feature (δ, ϕ). Then we can test for the presence of feature interaction in the following forms:

- Type 1: $(P + \delta_1) + \delta_2 \not\models \phi_2$:
  Earlier feature breaks later one.
- Type 2: $(P + \delta_1) + \delta_2 \not\models \phi_1$:
  Later feature breaks earlier one.
- Type 3: $P, P + \delta_1, P + \delta_2 \models \psi$ but $(P + \delta_1) + \delta_2 \not\models \psi$:
  (where ψ is a property of the base system.)
  Features combine to break system.
- Type 4: $\exists \phi.(P + \delta_1) + \delta_2 \models \phi$ but $(P + \delta_2) + \delta_1 \not\models \phi$:
  (where ϕ is a property of P, δ_1 or δ_2)
  Features do not commute.

Details of the results of the case study may be found in [8].

Superimposition. Our concept of feature construct is similar to the notion of *superimposition* [4]. A superimposition is a syntactic device for adding extra code to a given program, usually to make it better behaved with respect to other concurrently running programs. In the classic example of superimposition, extra code is added to enable processes to respond to interrogations from a supervisory process about whether they are awaiting further input, and this enables smooth termination of the system.

The superimposition construct proposed in [4] is suited to imperative languages, and therefore cannot be used directly for SMV. In imperative languages data and control flow are explicit, and the superimposition construct works by modifying them. For a declarative language like SMV data and control flow are implicit.

2 The feature construct for SMV

In our other papers [8, 9], we introduced two ways of introducing a feature in a SMV program: using *impose* and *treat*. Impose allows us to conditionally override the value of a variable with a new value. The syntax of impose features is

$$\texttt{if } \phi \texttt{ then impose next(x) := } f.$$

The intuitive meaning is that, whenever ϕ is true in a state, the value of x in the next state will be whatever the expression f evaluates to.

Treat allows us to behave *as if* a variable had a certain value, disregarding the actual value that the variable has. It thus introduces a mask on the variable. The syntax is:

$$\texttt{if } \phi \texttt{ then treat x = } f$$

Intuitively, when ϕ is true and the program reads the value of x, it gets the value of the expression f instead.

Clearly, impose and treat are in some sense dual of each other: impose affects the way a variable is written to, while treat affects the way it is read. This duality will be seen precisely in the semantics.

The tool SFI complies the feature construct into pure SMV as follows.

- For features of the form `if` ϕ `then impose next(`x`) :=` f:
  In assignments `next(`x`) :=` *oldexpr*, replace *oldexpr* by

  $$\begin{array}{l}\texttt{case}\\ \quad \phi \ : \ f;\\ \quad 1 \ : \ \mathit{oldexpr};\\ \texttt{esac}\end{array}$$

- For features of the form `if` ϕ `then treat` x `=` f:
  We rewrite f as f_1 union f_2 union ... union f_n such that each f_i is deterministic (see lemma 3.11). For each assignment `next(`x`) :=` e, we replace e by e_1 union ... union e_n where e_i is e with x replaced by

  $$\begin{array}{l}\texttt{case}\\ \quad \phi : \ f_i;\\ \quad 1 : \ x;\\ \texttt{esac}\end{array}$$

  This complicated procedure is equivalent, iff f is a deterministic expression, to the following simpler one: replace all occurrences of x in expressions by

```
case
     φ : f;
     1 : x;
esac
```

Whenever x is read, the value returned is not x's value, but the value of this expression. The reason for adopting the more complicated procedure will be clear in section 4.

Given an SMV program P and a feature δ, we write $P + \delta$ for the result of integrating δ into P in this way.

3 The semantics of SMV

Our aim in this paper is to understand when we can guarantee that certain classes of features written with our construct will not interact. To do this, we need first to analyse the theoretical properties of the feature construct. We explore this in Section 4. To set the stage for this, we have developed a semantics for SMV which we now describe. Our semantics are quite different in character (more denotational) than that given in [7]. Our semantics make it easier to deal with the "next" operator on the right hand side of assignments, a feature which SMV supports for a large class of programs, but which Ken McMillan does not cover in his semantics. A further extension (which is not supported by CMU SMV) is that we can allow non-deterministic expressions as conditions in "case" statements.

3.1 The syntax of SMV

Before we define the semantics of SMV, let us briefly review the syntax. We assume that only one module is defined (since, anyway, in the synchronous case the SMV model checker flattens a multi-module system to a single large module). An SMV program then consists of variable declarations, "x: *type*", and assignments, "next$(x) := e$" and "init$(x) := e$". The latter kind of assignment serves to define the set of initial states of the resulting automaton.

Types are (essentially) finite sets of values with certain operations on them. Expressions take the form

$$e \quad ::= \quad c \mid x \mid \mathrm{next}(x) \mid e_1 \circ e_2 \mid e_1 \text{ union } e_2 \mid \begin{array}{l} \text{case} \\ \quad ce_1 : e_1; \\ \quad ce_2 : e_2; \\ \quad \vdots \\ \quad ce_n : e_n; \\ \text{esac} \end{array}$$

where c is a constant, x a variable, and ce_i is a conditional expression (i.e. an expression of boolean type). We often write next(e) for the expression e with all variables $x_1, \ldots$ replaced by next$(x_1), \ldots$.

Definition 3.1 1. Let P be an SMV text consisting of a single module. Let n be the number of variables which occur in P, and $I = \{1, \ldots, n\}$. We will call the ith variable x_i.

2. Every type denotes a finite set. The type of a variable x_i is written $\mathrm{type}(x_i)$.

3. The set of states is $S = \prod_{i \in I} [\![\mathrm{type}(x_i)]\!]$.

4. If $s \in S$, we write $s(x_i)$ for the value of x_i in s, i.e. the ith component of s.

Let e be an expression in SMV. Its denotation $[\![e]\!]$ is a function in $S \times S \to \mathcal{P}(\mathrm{type}(e))$. Applying $[\![e]\!]$ to (s, s') returns the set of values that e could evaluate to if the current state is s and the next state is s'; note that, because e may refer to next-state variables as well as current-state variables, both the current state and the next state are required to evaluate it. The result of $[\![e]\!](s, s')$ is a set, because the expression e is (in general) non-deterministic.

Definition 3.2 The denotation of expressions is defined as follows, where $e_1, e_2, \ldots$ are expressions, $ce_1, \ldots$ are boolean expressions and $\circ$ any binary operator (such as $+, -, *, \&, \ldots$):

1. If d is a constant, then $[\![d]\!] = \lambda s s'.\, \{d\}$.

2. If x is a variable, then $[\![x]\!] = \lambda s s'.\, \{s(x)\}$, and $[\![\mathrm{next}(x)]\!] = \lambda s s'.\, \{s'(x)\}$.

3. $[\![e_1 \circ e_2]\!] = \lambda s s'.\, \left\{ v_1 \; [\![\circ]\!] \; v_2 \;\middle|\; v_1 \in [\![e_1]\!](s, s'),\; v_2 \in [\![e_2]\!](s, s') \right\}$,
   where $\circ$ is one of the operations $+, -, *, \&, \ldots$.

4. $[\![e_1 \text{ union } e_2]\!] = \lambda s s'.\, \Big([\![e_1]\!](s, s') \cup [\![e_2]\!](s, s') \Big)$;
   note that union in SMV denotes non-deterministic choice, and the expression $\{1, 2, 3\}$ is just shorthand for 1 union 2 union 3.

5. $$
\begin{aligned}
[\![\; & \text{case} \\
& \quad ce_1 : e_1; \\
& \quad ce_2 : e_2; \\
& \quad \vdots \\
& \quad ce_n : e_n; \\
& \text{esac} \;]\!] \\
= \lambda s s'.\, \Big(& \left\{ v \;\middle|\; 1 \in [\![ce_1]\!](s, s'),\; v \in [\![e_1]\!](s, s') \right\} \\
& \cup \left\{ v \;\middle|\; 0 \in [\![ce_1]\!](s, s'),\; 1 \in [\![ce_2]\!](s, s'),\; v \in [\![e_2]\!](s, s') \right\} \\
& \cup \left\{ v \;\middle|\; 0 \in [\![ce_1]\!](s, s') \cap [\![ce_2]\!](s, s'),\; 1 \in [\![ce_3]\!](s, s'),\; v \in [\![e_3]\!](s, s') \right\} \\
& \vdots \\
& \cup \left\{ 1 \;\middle|\; 0 \in \textstyle\bigcap_{i=1}^{n} [\![ce_i]\!](s, s') \right\} \quad \Big)
\end{aligned}
$$

Recall that in SMV, 0 denotes false and 1 denotes true; therefore, $0 \in [\![ce_3]\!](s, s')$ means that ce_3 can evaluate to false in (s, s'), etc. The last set in this union reflects the fact that if all the conditions ce_i evaluate to false, the case expression is defined to evaluate to 1.

When an expression e does not contain next(), we often write $[\![e]\!](s)$ rather than $[\![e]\!](s, s')$ to emphasise that $[\![e]\!]$ depends only on the current state.

Example 3.3 The expression

$$\begin{array}{l} \text{case} \\ \quad b : a + 1; \\ \quad 1 : a - 1; \\ \text{esac} \end{array}$$

denotes

$$\begin{aligned} \lambda s s'.\ \Big(&\Big\{v \;\Big|\; 1 \in [\![b]\!](s, s'),\ v \in [\![a+1]\!](s, s')\Big\} \\ &\cup \Big\{v \;\Big|\; 0 \in [\![b]\!](s, s'),\ 1 \in [\![1]\!](s, s'),\ v \in [\![a-1]\!](s, s')\Big\}\Big). \end{aligned}$$

If a and b are variables (as opposed to other kinds of expressions), then they evaluate deterministically and we obtain

$$\lambda s s'.\ \begin{cases} s(a) + 1 & \text{if } s(b) \\ s(a) - 1 & \text{otherwise} \end{cases}$$

The semantics of expressions is thus quite straightforward. However, it fails an important property of substitutivity. We might expect that if two expressions e_1, e_2 denote the same thing, and we substitute for the variable x a third expression e, the resulting expressions should also denote the same thing. Let $e_1[e/x]$ mean the expression e_1 with all occurrences of the variable x (not within next()) replaced by the expression e, and $e_1[e/\text{next}(x)]$ is e_1 with all occurrences of $\text{next}(x)$ replaced by e. Note that we can never get nested next()s by performing these substitutions (i.e. the set of expressions is closed under them).

Remark 3.4 (Substitutivity) $[\![e_1]\!] = [\![e_2]\!]$ does not imply $[\![e_1[e/x]]\!] = [\![e_2[e/x]]\!]$.

Example 3.5 Here is an example of the failure of substitutivity:

$$\begin{aligned} e_1 &= 2 * x \\ e_2 &= x + x \\ e &= \{2, 3\} \\ [\![e_1[e/x]]\!] &= [\![2 * \{2, 3\}]\!] \\ &= \lambda s s'.\ \{4, 6\} \\ [\![e_2[e/x]]\!] &= [\![\{2, 3\} + \{2, 3\}]\!] \\ &= \lambda s s'.\ \{4, 5, 6\} \end{aligned}$$

The reason for the failure is clear: after substituting in $x+x$, the non-deterministic expression will occur twice and can evaluate differently the two times. In $2*x$ it occurs only once.

Substitutivity holds if e is deterministic, or e_1, e_2 have just one occurrence of x or $\mathrm{next}(x)$. To prove this, we need the following lemmas. Write s_x^v for the state just like s except that x has the value v.

Lemma 3.6 Let e, f be SMV expressions, then

$$[\![e[f/x]]\!](s,s') \quad \supseteq \quad \bigcup_{v \in [\![f]\!](s,s')} [\![e]\!](s_x^v, s')$$

Moreover, equality holds if f is deterministic, or e has just one occurrence of x.

Proof We prove the lemma by induction on the structure of e. We give only one case:

- $e = e_1 \circ e_2$.

  $$\begin{aligned}
  &[\![(e_1 \circ e_2)[f/x]]\!](s,s') \\
  &= \left\{ w_1 \; [\![\circ]\!] \; w_2 \;\middle|\; w_1 \in \underbrace{[\![e_1[f/x]]\!](s,s')}_{\supseteq \{w \mid w \in [\![e_1]\!](s_x^v, s'),\ v \in [\![f]\!](s,s')\},\text{ by Ind.Hyp.}},\ w_2 \in [\![e_2[f/x]]\!](s,s') \right\} \text{ by semantics of } \circ \\
  &\supseteq \left\{ w_1 \; [\![\circ]\!] \; w_2 \;\middle|\; w_i \in [\![e_i]\!](s_x^{v_i}, s'),\ v_i \in [\![f]\!](s,s'),\ i = 1,2 \right\}, \text{ Ind.Hyp.} \\
  &\supseteq \left\{ w_1 \; [\![\circ]\!] \; w_2 \;\middle|\; w_i \in [\![e_i]\!](s_x^{v}, s'),\ v \in [\![f]\!](s,s'),\ i = 1,2 \right\}, \text{ rules of sets} \\
  &= \left\{ [\![e_1 \circ e_2]\!](s_x^v, s') \;\middle|\; v \in [\![f]\!](s,s') \right\}, \text{ def. of } \circ
  \end{aligned}$$

  Now suppose f is deterministic. Then the first two occurrences of $\supseteq$ above become $=$ by Ind.Hyp., and the third becomes $=$ by the fact that $[\![f]\!](s)$ is a singleton.

  Suppose e has just one occurrence of x, say, in e_1. Then e_2 does not depend on x. By inductive hypothesis, $[\![e_1[f/x]]\!](s,s') = \bigcup [\![e_1]\!](s_x^v, s')$ (first $\supseteq$). Also $[\![e_2[f/x]]\!](s,s') = \bigcup [\![e_2]\!](s_x^v, s') = [\![e_2]\!](s,s')$, i.e. $[\![e_2]\!](s_x^v, s')$ is independent of the choice of $v \in [\![f]\!](s,s')$, so the middle line becomes $\{w_1 \; [\![\circ]\!] \; w_2 \mid w_1 \in [\![e_1]\!](s_x^{v_1}, s'),\ w_2 \in [\![e_2]\!](s,s'),\ v_1 \in [\![f]\!](s,s')\}$. This justifies $=$ for the second and third $\supseteq$.

$\square$

We can prove a similar lemma for substitutions on $\mathrm{next}(x)$.

Lemma 3.7 Let e, f be SMV expressions, then

$$[\![e[f/\mathrm{next}(x)]]\!](s,s') \quad \supseteq \quad \bigcup_{v \in [\![f]\!](s,s')} [\![e]\!](s, {s'}_x^v)$$

Moreover, equality holds if f is deterministic, or e has just one occurrence of $\mathrm{next}(x)$.

(Proofs not given here can be found in the long version of the paper, available as a technical report.)

Corollary 3.8 Let e, f be SMV expressions. If f does not contain $\mathrm{next}(x)$, then

$$[\![(e[f/x])[\mathrm{next}(f)/\mathrm{next}(x)]]\!] \supseteq \bigcup_{\substack{v \in [\![f]\!](s,s') \\ v' \in [\![f]\!](s,s')}} [\![e]\!](s_x^v, {s'}_x^{v'})$$

Example 3.9 We give an example of proper inclusion for Lemma 3.6. Again let $e = x+x$ and $f = \{2,3\}$. Note f is non-deterministic.
LHS $= [\![(x+x)[\{2,3\}/x]]\!](s,s') = [\![\{2,3\}+\{2,3\}]\!](s,s') = \{4,5,6\}$.
RHS $= \bigcup_{v\in\{2,3\}} [\![x+x]\!](s_x^v, s') = \{4,6\}$.

Corollary 3.10 If e is deterministic, or e_1, e_2 have just one occurrence of x, then $[\![e_1]\!] = [\![e_2]\!]$ implies $[\![e_1[e/x]]\!] = [\![e_2[e/x]]\!]$.

The previous lemmas and examples show that the SMV language of non-deterministic expressions, although simple and intuitive, fails a property of substitutivity. This property is important to us because the *treat* feature is defined in terms of substitution, and we would like the property in order to guarantee that the treat feature is nicely behaved.

Our first approach to this problem was to restrict to the cases of lemmas 3.6 and 3.7 in which f is deterministic. Looking again at the definition of the feature construct in section 2, this would mean that f and ϕ in the treat feature would have to be deterministic.

We can avoid this restriction, however, by defining substitution in a cleverer way. First note that all the non-determinism in f can be expressed at the outermost level.

Lemma 3.11 Let f be a (possibly non-deterministic) expression. Then there are deterministic expressions f_1, f_2, ... , f_n such that

$$[\![f]\!] = [\![f_1 \text{ union } f_2 \text{ union} \ldots \text{union } f_n]\!].$$

Proof Induction on the structure of f. (Rewriting the expression is purely mechanical.) $\square$

The new operator uses this way of rewriting expressions.

Definition 3.12 Let e, f be expressions and x a variable. The expression $e\{f/x\}$ is defined thus:

$$e\{f/x\} = e[f_1/x] \text{ union } e[f_2/x] \text{ union} \ldots \text{union } e[f_n/x]$$

where f has been written f_1 union f_2 union ... union f_n with each f_i deterministic (see lemma).

Obtaining $e\{f/x\}$ is easily automated, since rewriting f according to Lemma 3.11 is a straightforward syntactic manipulation.

Remark 3.13 Let e, f_1, f_2 be SMV expressions. If f_1 does not contain next(x) and x does not occur in f_2 then $(e\{f_1/x\})\{f_2/\text{next}(x)\} = (e\{f_2/\text{next}(x)\})\{f_1/x\}$ up to reordering of subterms. This also holds for ordinary substitution $\cdot[\cdot]$. In the remainder of this paper, we will usually have $f_2 = \text{next}(f_1)$, in which case the remark applies.

With this new operator, we obtain the desired result for substitution:

Lemma 3.14 Let e, f be SMV expressions. Then

$$[\![e\{f/x\}]\!](s,s') = \bigcup_{v\in[\![f]\!](s,s')} [\![e]\!](s_x^v, s')$$

and

$$[\![e\{f/\text{next}(x)\}]\!](s,s') = \bigcup_{v\in[\![f]\!](s,s')} [\![e]\!](s, {s'}_x^v)$$

Proof By induction, using lemmas. □

The other side of substitutivity asks that $[\![e_1]\!] = [\![e_2]\!]$ implies $[\![e[e_1/x]]\!] = [\![e[e_2/x]]\!]$, i.e., substituting equivalent expressions into an expression results in equivalent expressions. (Analogously for $e[e_1/\mathrm{next}(x)]$.) This holds without qualification in our semantics:

Proposition 3.15 $[\![e_1]\!] = [\![e_2]\!]$ implies $[\![e[e_1/x]]\!] = [\![e[e_2/x]]\!]$ and $[\![e[e_1/\mathrm{next}(x)]]\!] = [\![e[e_2/\mathrm{next}(x)]]\!]$.

Proof Induction on the structure of e. If e is a constant a variable, or $\mathrm{next}(z)$ for a variable z, the result is straightforward; otherwise,

- if $e = f_1 \circ f_2$ then

$$\begin{aligned}
[\![e[e_1/x]]\!](s,s') &= [\![(f_1 \circ f_2)[e_1/x]]\!](s,s') \\
&= [\![f_1[e_1/x] \circ f_2[e_1/x]]\!](s,s') \\
&= [\![f_1[e_1/x]]\!](s,s')\ [\![\circ]\!]\ [\![f_2[e_1/x]]\!](s,s') \\
&= [\![f_1[e_2/x]]\!](s,s')\ [\![\circ]\!]\ [\![f_2[e_2/x]]\!](s,s')
\end{aligned}$$

  The last step uses the inductive hypothesis. Now this expression can be packed up again to obtain $[\![e[e_2/x]]\!](s,s')$.

- Union, case statements, etc: similar. □

Substitutivity will be important for the application to features in a later section.

We return to the main theme of this section, which is defining the semantics of SMV. Having examined the semantics of expressions, we now give the semantics of complete programs. Since expressions do most of the work in SMV programs, there is not much more to do:

Definition 3.16 Assignments denote relations on $S \times S$:

- $[\![\mathrm{next}(x) := e]\!] = \{(s,s') \mid s'(x) \in [\![e]\!](s,s')\}$;
- $[\![x := e]\!] = \{(s,s') \mid s(x) \in [\![e]\!](s,s')\}$.

The transition relation is given by

$$R = \bigcap_{a \text{ an assignment}} [\![a]\!].$$

An SMV program P denotes a pair $[\![P]\!] = (S, R)$, where S is the set of states (given in definition 3.1, and R is the transition relation. We may now apply this semantics to verify the examples given in section 2.

4 Semantics of the feature construct

The results of our previous papers are entirely experimental. In this section, we aim to apply the semantics of SMV developed so far to features, and thus to provide some theoretical results about the feature construct. More precisely, we wish to find out:

- When (if ever) can we guarantee that two features will commute with each other?
- To what extent does the meaning of a feature depend on syntactical details of the program with which it is integrated?

Answering questions such as these will put us in a better position to assess the usefulness of the feature construct idea.

Definition 4.1 (Admissible SMV programs) An SMV program is *admissible* if:

- there are no assignments to current variables;
- in any assignment of the form `next(x) :=` e, `next(x)` does not occur in e.

We assume that the base system is an admissible SMV program, and we also make the assumption that, in the features

`if` ϕ `then impose next(x) :=` f
`if` ϕ `then treat x =` f

the condition ϕ is deterministic and does not contain next(), and that next(x) does not occur in the expression f.

As stated before, we do not allow next() to occur in treat features: if, in the program, an expression referred to next(x), then the integration of such a feature would lead to double nexts, i.e. a reference to a successor state of the next state, which cannot be determined from the current state. This restriction also means that for treat features we can write $[\![f]\!](s)$ instead of $[\![f]\!](s,s')$, since f cannot depend on s'.

For some proofs we will assume that the expression f occurring in the feature is deterministic. In that case $[\![f]\!](s)$ is a singleton for any s. We will write $[\![f]\!](s)$ to be the *value* of f in s, rather than the singleton set containing that value, in order to simplify notation. It means we can write $s_x^{[\![f]\!](s)}$ for the state like s but with x having the value of f in s, etc.

For ease of description, we use the symbol δ for features using the keyword *impose*, and ϵ for features using the keyword *treat*.

Definition 4.2 (Semantics of features) Let A be an automaton, i.e. a binary relation on a set of states.

- If δ is the feature

  `if` ϕ `then impose next(x) :=` f

  then

$$\begin{aligned}[\![\delta]\!](A) \;=\; & \Big\{(s,s') \;\Big|\; s \not\Vdash \phi,\ (s,s')\in A\Big\} \\ & \cup\Big\{(s,{s'}_x^v) \;\Big|\; s \Vdash \phi,\ (s,s')\in A,\ v\in[\![f]\!](s,s')\Big\}.\end{aligned}$$

  Thus, we retain transitions $(s,s')\in A$ which do not trigger the feature ($s \not\Vdash \phi$); in the case that the feature is triggered ($s \Vdash \phi$) we alter the target state to take account of the impose.

- If ϵ is the feature

  `if` ϕ `then treat x =` f

  then

$$
\begin{aligned}
[\![\epsilon]\!](A) \;=\; & \Big\{(s,s') \;\Big|\; s \not\Vdash \phi,\, s' \not\Vdash \phi,\, (s,s') \in A\Big\} \cup \\
& \Big\{(s,s') \;\Big|\; s \Vdash \phi,\, s' \not\Vdash \phi,\, \exists v \in [\![f]\!](s).\, (s^v_x, s') \in A\Big\} \cup \\
& \Big\{(s,s') \;\Big|\; s \not\Vdash \phi,\, s' \Vdash \phi,\, \exists v \in [\![f]\!](s').\, (s, s'^v_x) \in A\Big\} \cup \\
& \Big\{(s,s') \;\Big|\; s \Vdash \phi,\, s' \Vdash \phi,\, \exists v \in [\![f]\!](s), v' \in [\![f]\!](s').\, (s^v_x, s'^{v'}_x) \in A\Big\}
\end{aligned}
$$

  Again, we retain transitions $(s,s') \in A$ which do not trigger the feature. Here, if the feature is triggered, we behave as if x had the value of f in the current or next state, respectively. That is, we transition from s to s' if there was a transition from $s^{[\![f]\!](s)}_x$ to $s'^{[\![f]\!](s)}_x$. (Recall that we ruled out occurrences of next() in f.)

Remark 4.3 The feature

`if` ϕ `then treat x =` f

is equivalent to the feature

```
treat x = case
            φ: f;
            1: x;
          esac
```

where we have omitted "if true then" for obvious reasons. The equivalent reformulation is not as intuitive to the programmer, but it will help simplify some of the mathematical proofs.

Remark 4.4 If the expression d in the treat feature $\epsilon =$ "`if` ϕ `then treat` x `=` d" is deterministic and we rewrite ϵ according to remark 4.3, the semantics for ϵ simplify to

$$[\![\epsilon]\!](A) = \Big\{(s,s') \;\Big|\; (s^{[\![f]\!](s)}_x, s'^{[\![f]\!](s')}_x) \in A\Big\}.$$

where f stands for `case` $\phi : d$`; 1:` x`; esac`.

Our aim in this section is to show that the semantics of features given above coincides with what SFI actually does. As indicated above, we write $P + \delta$ for the result of integrating δ into P. Thus, we aim to prove:

Lemma 4.5 Let δ and ϵ be as above, let P be an admissible SMV program.

1. If P is deadlock free[2] and next(x) does not occur in f, then $[\![P + \delta]\!] = [\![\delta]\!]([\![P]\!])$.

2. If f and ϕ are deterministic and contain no occurrences of x and of the next() operator, then $[\![P + \epsilon]\!] = [\![\epsilon]\!]([\![P]\!])$.

[2] i.e., from each state in the denoted transition system there is at least one successor state.

Proof 1. We show $(s,s') \in [\![P+\delta]\!] \Leftrightarrow (s,s') \in [\![\delta]\!]([\![P]\!])$.

Suppose $s \not\Vdash \phi$. Then

$$\begin{aligned}(s,s') \in [\![P+\delta]\!] &\Leftrightarrow (s,s') \in [\![P]\!] && \text{by construction of } P+\delta\\ &\Leftrightarrow (s,s') \in [\![\delta]\!](P) && \text{by def. of } [\![\delta]\!]([\![P]\!])\end{aligned}$$

Suppose $s \Vdash \phi$, and suppose the assignment to `next(x)` is `next(x)` $:= e$. Let P' be P without this assignment to next(x). Notice that $s''^{[\![f]\!](s,s')}_x = s''^{[\![f]\!](s,s'')}_x$, since s' and s'' differ only in their value for x but next(x) does not occur in f.

$$(s,s') \in [\![P+\delta]\!]$$

$$\Leftrightarrow (s,s') \in [\![P']\!] \wedge s'(x) \in [\![f]\!](s,s') \qquad \text{by construction of } P+\delta$$

$$\Leftrightarrow \exists s'', f_i \in \det(f).\, s' = s''^{[\![f_i]\!](s,s')}_x \wedge (s,s'') \in [\![P]\!] \qquad \text{(see next paragraph)}$$

$$\Leftrightarrow (s,s') \in [\![\delta]\!]([\![P]\!]) \qquad \text{def. of } [\![\delta]\!]$$

The middle equivalence is justified as follows:
$\Rightarrow$. Since $[\![P]\!]$ is deadlock free, there is a successor state s'' of s. Let $s'' = s'^v_x$ for some $v \in [\![e]\!](s)$. Then, for some $f_i \in \det(f)$, $s''^{[\![f_i]\!](s,s')}_x = s'$ since $s'(x) \in [\![f]\!](s,s')$. We have $(s,s'') \in [\![P']\!]$. To show $(s,s'') \in [\![P]\!]$ it is sufficient to show that it satisfies next$(x) = e$.
$\Leftarrow$. Suppose s'' is as given. We easily obtain that $(s,s') \in [\![P']\!]$ and $s'(x) \in [\![f]\!](s,s')$.

2. By remarks 4.3 and 4.4 we can assume that ϵ has the form `treat` x `=` f. We know that f is deterministic and contains no x or next(). Given an assignment next(x) $:= e$, these restrictions ensure that ϵ will not introduce a circular dependency in this assignment.[3]
We first require the following
Lemma: If f is a deterministic expression, $[\![\epsilon]\!](A \cap B) = [\![\epsilon]\!](A) \cap [\![\epsilon]\!](B)$.
Proof of lemma:
$$\begin{aligned}&(s,s') \in [\![\epsilon]\!](A) \cap [\![\epsilon]\!](B)\\ \Leftrightarrow\ &(\exists v \in [\![f]\!](s), v' \in [\![f]\!](s').\, (s^v_x, s'^{v'}_x) \in A) \wedge\\ &(\exists v \in [\![f]\!](s), v' \in [\![f]\!](s').\, (s^v_x, s'^{v'}_x) \in B)\\ \Leftrightarrow\ &(\exists v \in [\![f]\!](s), v' \in [\![f]\!](s').\, (s^v_x, s'^{v'}_x) \in A \cap B)\\ \Leftrightarrow\ &(s,s') \in [\![\epsilon]\!](A \cap B)\end{aligned}$$
The second step relies on $[\![f]\!](s)$ being a singleton, i.e. on the determinism of f.

Now think of P as the set of its assignments. For each $a \in P$ we prove $[\![a+\epsilon]\!] = [\![\epsilon]\!]([\![a]\!])$. Then, by definition of $[\![P]\!]$ and the lemma above,

$$[\![\epsilon]\!]([\![P]\!]) = [\![\epsilon]\!]\Big(\bigcap_{a\in P}[\![a]\!]\Big) = \bigcap_{a\in P}[\![\epsilon]\!]([\![a]\!]) = \bigcap_{a\in P}[\![a+\epsilon]\!] = [\![P+\epsilon]\!].$$

Let a be the assignment `next(y) :=` e. We prove $[\![a+\epsilon]\!] = [\![\epsilon]\!]([\![a]\!])$.

[3] In fact, we can only prevent circular dependencies within *one* assignment. The next values of other variables may depend on next(x), and vice versa; in such a case, the circular dependencies will extend over more than one assignment.

This equality holds irrespective of whether f is deterministic or not. For simplicity, however, we will make use of our hypothesis that f is deterministic and f contains no next() operator.

$$
\begin{aligned}
&[\![a+\epsilon]\!] \\
&\quad = [\![\text{next}(y) := e\,[f/x]\,[\text{next}(f)/\text{next}(x)]]\!] && \text{def. of } +\epsilon \\
&\quad = \left\{(s,s') \;\middle|\; s'(y) \in [\![e\,[f/x]\,[\text{next}(f)/\text{next}(x)]]\!](s,s')\right\} && \text{(assignment)} \\
&\quad = \left\{(s,s') \;\middle|\; s'(y) \in [\![e]\!]\left(s_x^{[\![f]\!](s)}, {s'}_x^{[\![f]\!](s')}\right)\right\} && \text{(corollary 3.8)} \\
&\quad = \left\{(s,s') \;\middle|\; (s_x^{[\![f]\!](s)}, {s'}_x^{[\![f]\!](s')}) \in [\![a]\!]\right\} && \text{(assignment)} \\
&\quad = [\![\epsilon]\!]([\![a]\!]) && \text{(remark 4.4)}
\end{aligned}
$$

□

Theorem 4.6 The feature constructs are syntax-invariant. Let P_1, P_2 be programs and η a feature (could be δ-type or ϵ-type). Then

$$[\![P_1]\!] = [\![P_2]\!] \text{ implies } [\![P_1 + \eta]\!] = [\![P_2 + \eta]\!].$$

Proof Immediate corollary of the lemma. □

Note, however, that $[\![P_1]\!] = [\![P_2]\!]$ is rather strong: it says P_1, P_2 denote the same transition system, even on the non-reachable part.

4.1 Properties of the feature construct

Theorem 4.7 (Idempotence of feature addition)

1. The assignment to next(x) in P is next$(x) := e$, and both e and f and ϕ in the feature δ do not contain next(x), then $[\![P+\delta+\delta]\!] = [\![P+\delta]\!]$.

2. If x does not occur in the expressions ϕ and f, and ϕ and f do not contain next(), then $[\![P+\epsilon+\epsilon]\!] = [\![P+\epsilon]\!]$.

Proof 1. Similar to what we did with treat features, we rewrite δ to "impose next$(x) := c$", where c denotes the expression

```
case
    φ : f;
    1 : e;
esac
```

We prove that $[\![\delta]\!]([\![\delta]\!]([\![P]\!])) = [\![\delta]\!]([\![P]\!])$.
Since c does not mention next(x) we know that $[\![c]\!](s,s') = [\![c]\!](s,{s'}_x^v)$ for any $v \in \text{type}(x)$, thus

$$\begin{aligned}(s,s') \in [\![\delta]\!]([\![\delta]\!]([\![P]\!])) &\Leftrightarrow s' = s''^{[\![c]\!](s,s'')}_x,\ (s,s'') \in [\![\delta]\!](P)\\ &\Leftrightarrow s' = s''^{[\![c]\!](s,s'')}_x,\ s'' = s'''^{[\![c]\!](s,s''')}_x,\ (s,s''') \in P\\ &\Leftrightarrow s' = s'''^{[\![c]\!](s,s''')}_x,\ (s,s''') \in P\\ &\Leftrightarrow (s,s') \in [\![\delta]\!](P)\end{aligned}$$

2. Again we assume that ϵ has the form "treat $x = f$". Let $A = [\![P]\!]$. We prove that $\epsilon(\epsilon(A)) = \epsilon(A)$. We write $\tilde{s}$ and $\tilde{s}'$ as a shorthand for $s^{[\![f]\!](s,s')}_x$ and $s'^{[\![f]\!](s,s')}_x$, respectively.

$$\begin{aligned}(s,s') \in \epsilon(\epsilon(A)) &\Leftrightarrow (\tilde{s},\tilde{s}') \in \epsilon(A)\\ &\Leftrightarrow \left(\tilde{s}^{[\![f]\!](\tilde{s},\tilde{s}')}_x, (\tilde{s}')^{[\![f]\!](\tilde{s},\tilde{s}')}_x\right) \in A)\\ &\Leftrightarrow (s^{[\![f]\!](\tilde{s},\tilde{s}')}_x, s'^{[\![f]\!](\tilde{s},\tilde{s}')}_x) \in A)\\ &\Leftrightarrow (s^{[\![f]\!](s,s')}_x, s'^{[\![f]\!](s,s')}_x) \in A)\\ &\Leftrightarrow (s,s') \in \epsilon(A)\end{aligned}$$

The first and second equivalences are obtained by rewriting; the third and the fourth exploit the fact that x and $\mathrm{next}(x)$ do not occur in f. □

Finally, let us look at when features commute with each other. In general we do not expect that features should commute. However, when they do, it implies a strong form of non-interaction.

Consider the families of features

$$\begin{aligned}\delta_i =& \texttt{if}\ \phi_i\ \texttt{then impose next}(x_i) := f_i\\ \epsilon_i =& \texttt{if}\ \phi_i\ \texttt{then treat}\ x_i = f_i\end{aligned}$$

We explore when δ_1 commutes with δ_2, etc.

As usual we rule out features that may lead to circular assignments, i.e. for impose features, f_i must not refer to $\mathrm{next}(x_i)$, and for treat features, f_i must not refer to x_i or use $\mathrm{next}()$. Also, for both types of features, ϕ_i must not contain $\mathrm{next}()$.

Theorem 4.8

1. $P + \delta_1 + \delta_2 = P + \delta_2 + \delta_1$, if x_1, x_2 are distinct variables and δ_1 does not use $next(x_2)$ and vice versa.

2. $P + \delta_1 + \epsilon_2 = P + \epsilon_2 + \delta_1$ if x_2 does not occur in ϕ_1, x_1 and x_2 are distinct variables, and x_1 does not occur in f_2 or ϕ_2.

3. $P + \epsilon_1 + \epsilon_2 = P + \epsilon_2 + \epsilon_1$ if:
   - x_1, x_2 are distinct variables, and
   - x_1 does not occur in ϕ_2 or f_2, and
   - x_2 does not occur in ϕ_1 or f_1;

Proof For the proof we again assume the simple form of treat features.
Note that

$$(s,t) \in [\![\delta_i]\!](A) \Leftrightarrow \left[\begin{array}{l} s \not\Vdash \phi_i,\ (s,t) \in A\\ s \Vdash \phi_i,\ t = t'^{[\![f_i]\!](s,t')}_{x_i},\ (s,t') \in A\end{array}\right]$$

where we use the notation: in square brackets, comma means and, and vertical juxtaposition means or; and

$$(s,t) \in [\![\epsilon_i]\!](A) \Leftrightarrow (s^{[\![f_i]\!](s)}_{x_i}, t^{[\![f_i]\!](t)}_{x_i}) \in A$$

1. Expanding $[\![\delta_1]\!]([\![\delta_2]\!](A))$, we see that

$$(s,s') \in [\![\delta_1]\!]([\![\delta_2]\!](A)) \Leftrightarrow$$
$$\left[\begin{array}{ll} s \Vdash \neg\phi_1 \wedge \neg\phi_2, & (s,s') \in A \\ s \Vdash \neg\phi_1 \wedge \phi_2, & s = t_{x_2}^{[\![f_2]\!](s,t)},\ (s,t') \in A \\ s \Vdash \phi_1 \wedge \neg\phi_2, & s = t_{x_1}^{[\![f_1]\!](s,t)},\ (s,t') \in A \\ s \Vdash \phi_1 \wedge \phi_2, & s = (t_{x_2}^{[\![f_2]\!](s,t)})_{x_1}^{[\![f_1]\!](s,t_{x_2}^{[\![f_2]\!](s,t)})},\ (s,t') \in A \end{array}\right]$$

If x_1 and x_2 are distinct, and $[\![f_1]\!](s,t)$ does not depend on $t(x_2)$ and, symmetrically, $[\![f_2]\!](s,t)$ is independent of $t(x_1)$, then

$$(t_{x_2}^{[\![f_2]\!](s,t)})_{x_1}^{[\![f_1]\!](s,t_{x_2}^{[\![f_2]\!](s,t)})} = (t_{x_2}^{[\![f_2]\!](s,t)})_{x_1}^{[\![f_1]\!](s,t)} = (t_{x_1}^{[\![f_1]\!](s,t)})_{x_2}^{[\![f_2]\!](s,t)} = (t_{x_1}^{[\![f_1]\!](s,t)})_{x_2}^{[\![f_2]\!](s,t_{x_1}^{[\![f_1]\!](s,t)})}$$

2. Expanding $[\![\delta_1]\!]([\![\epsilon_2]\!](A))$, obtain

$$(s,t) \in [\![\delta_1]\!]([\![\epsilon_2]\!](A)) \Leftrightarrow \left[\begin{array}{ll} s \Vdash \neg\phi_1, & (s_{x_2}^{[\![f_2]\!](s)}, t_{x_2}^{[\![f_2]\!](t)}) \in A \\ s \Vdash \phi_1, & t = t'^{[\![f_1]\!](s,t')}_{x_1},\ (s_{x_2}^{[\![f_2]\!](s)}, t'^{[\![f_2]\!](t')}_{x_2}) \in A \end{array}\right]$$

Expanding $[\![\epsilon_2]\!]([\![\delta_1]\!](A))$, we get

$$(s,t) \in [\![\epsilon_2]\!]([\![\delta_1]\!](A)) \Leftrightarrow \left[\begin{array}{ll} s \Vdash \neg\phi_1', & (s_{x_2}^{[\![f_2]\!](s)}, t_{x_2}^{[\![f_2]\!](t)}) \in A \\ s \Vdash \phi_1', & t_{x_2}^{[\![f_2]\!](t)} = t'^{[\![f_1]\!](s)}_{x_1},\ (s_{x_2}^{[\![f_2]\!](s)}, t') \in A \end{array}\right]$$

where ϕ_1' stands for $\phi_1[f_2/x_2][\mathrm{next}(f_2)/\mathrm{next}(x_2)]$.
ϕ_1 holds for the same states in both cases if x_2 does not occur in ϕ_1. Now, if $x_1 \neq x_2$ and x_1 does not occur in f_2, the last line is equivalent to

$$\phi_1[f_2/x_2],\ t = t'^{[\![f_1]\!](s)}_{x_1},\ (s_{x_2}^{[\![f_2]\!](s)}, t'^{[\![f_2]\!](t')}_{x_2}) \in A.$$

3. Expanding $[\![\epsilon_1]\!]([\![\epsilon_2]\!](A))$ and $[\![\epsilon_2]\!]([\![\epsilon_1]\!](A))$ we see that

$$(s,t) \in [\![\epsilon_1]\!]([\![\epsilon_2]\!](A)) \Leftrightarrow ((s_{x_1}^{[\![f_1]\!](s)})_{x_2}^{[\![f_2[f_1/x_1]]\!](s_{x_1}^{[\![f_1]\!](s)})}, (t_{x_1}^{[\![f_1]\!](t)})_{x_2}^{[\![f_2[f_1/x_1]]\!](t_{x_1}^{[\![f_1]\!](t)})}) \in A$$

and

$$(s,t) \in [\![\epsilon_2]\!]([\![\epsilon_1]\!](A)) \Leftrightarrow ((s_{x_2}^{[\![f_2]\!](s)})_{x_1}^{[\![f_1[f_2/x_2]]\!](s_{x_2}^{[\![f_2]\!](s)})}, (t_{x_2}^{[\![f_2]\!](t)})_{x_1}^{[\![f_1[f_2/x_2]]\!](t_{x_2}^{[\![f_2]\!](t)})}) \in A$$

Here we have used the the substitution lemma (lemma 3.8) in the form $[\![f_2]\!](s_{x_1}^{[\![f_1]\!](s)}) = [\![f_2[f_1/x_1]]\!](s)$.

Comparing $\epsilon_1(\epsilon_2(A))$ with $\epsilon_2(\epsilon_1(A))$, we see that they are equal provided the syntactic substitutions have no effect, i.e. there are no occurrences of x_1 in f_2, or of x_2 in f_1. The same condition also ensures that f_2 does not depend on x_1 and f_1 not on x_2, so that $[\![f_2[f_1/x_1]]\!](s_{x_2}^{[\![f_2]\!](s)}) = [\![f_2[f_1/x_1]]\!](s)) = [\![f_2]\!](s)$, and symmetrically.

$\square$

5 Conclusions

The experimental results of [8] are enhanced with theoretical results showing:

- that, with an appropriate notion of equivalence between SMV programs, features are insensitive to the syntax of the underlying program; and
- circumstances in which features are idempotent and commute.

These results will prove to be helpful to the model checking process. In our case study of the phone system, we quickly found that with the addition of features the system quickly grew too large to verify. The results in this paper suggest that some results can be obtained purely by analysis of the features, rather than by model checking the extended system. For example, theorems 4.7 and 4.8 allow us to reduce the number of feature combinations that need to be checked. In the future, we hope to show that it is sufficient to check the feature with an abstract version of the base system to prove a property of the full system with the feature.

References

[1] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems III*, Kyoto, Japan, 1995. Omsha Press.

[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a reimplementation of SMV. In B. Steffen and T. Margaria, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT-98)*, BRICS Notes Series, pages 25–31, Aalborg, 1998. Available from afrodite.itc.it:1024/~cimatti/.

[3] P. Dini et al., editors. *Feature Interactions in Telecommunications and Distributed Systems IV*, Montreal, Canada, June 1997. IOS Press.

[4] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[5] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press, September 1998.

[6] K. McMillan. The SMV language. Available from www-cad.eecs.berkeley.edu/~kenmcmil, June 1998.

[7] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[8] M. C. Plath and M. D. Ryan. Plug-and-play features. In Kimbler and Bouma [5], pages 150–164.

[9] M. C. Plath and M. D. Ryan. SFI: a feature integration tool. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 201–216. Springer, 1999.