Feature Integration as Substitution

Dimitar P. Guelev, Mark D. Ryan and Pierre Yves Schobbens

School of Computer Science, Birmingham University, UK
E-mail: {D.P.Guelev, M.D.Ryan}@cs.bham.ac.uk
Institut d'Informatique, Facultés Universitaires de Namur, Belgium
E-mail: pys@info.fundp.ac.be

Abstract. In this paper we analyse the development of automated systems by means of adding features to a basic system. Our approach is to describe systems in temporal logic. We regard the process of integrating features as a transformation of those temporal logic descriptions. We use substitution of predicates as the basic means to achieve feature readiness of descriptions and define feature integration. We show that several description formalisms for features known from the literature can be fitted into the formal framework of our analysis, despite the fact that they have been initially motivated by different observations. Among these formalisms are Samborski's *stack service model* [12], the *feature construct* for SMV [10] and others. We argue that the way to address the verification problems which are specific to systems with features provided by our analysis is clear, convenient and based on classical and well established logical notions only. The logic we use in examples of description in this paper is the duration calculus with higher order quantifiers and iteration [15, 8, 1, 5].

Introduction

New models of automated systems, such as computerised systems, and new versions of software are typically obtained by making additions and small other changes to earlier ones. Such additions are known as *features* [2, 4]. Systems which have been built by adding features are more likely to have design faults than others, because features are typically designed separately from basic systems and from each other. Nevertheless, the method of obtaining better designs by integrating features is immensely more efficient that its known alternatives. That is why verification is important for such systems. Telecommunication systems are the greatest source of examples of systems with features. Sets of features, whose combined effect on the behaviour of a basic system is unconventional, are said to have an *interaction* in the literature on telecommunications. Identifying feature interactions is one of the most outstanding verification problems related to the development of systems by incorporating features.

In general, a feature F can be regarded as a mapping $F : \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$ is the class of systems in question, such that given a system $S \in \mathcal{S}$ an (attempt to) integrate F into S results in a system $F(S)$. Less generally, but very commonly, a feature is a prefabricated item F, and the result of integrating F into a system S is then denoted by $S \oplus F$ where $\oplus$ stands for an operation of feature integration.

In this paper we are interested in systems' behaviour and therefore assume that the relevant properties of such behaviour are described in a temporal logic. We assume that there

is a *modelling relation* $\models$ on $\mathcal{S} \times \mathcal{L}(\mathcal{S})$ where $\mathcal{L}(\mathcal{S})$ stands for the language of the chosen temporal logic based on the vocabulary which corresponds to the class of systems $\mathcal{S}$ such that

$$S \models \varphi$$

denotes that *the logic model(s) of all the potential behaviours of S satisfy φ*, and in particular,

$$S \oplus F \models \varphi$$

stands for *S with feature F added to it satisfies φ*. Using $\models$ and $\oplus$, some interesting statements about systems with features can be formalised:

Feature interaction

$S \oplus F_1 \oplus \ldots \oplus F_n \not\models \varphi$	the "combined effort" of all the features breaks φ;
$S \oplus F_1 \oplus \ldots \oplus F_{i-1} \oplus F_{i+1} \oplus \ldots \oplus F_n \models \varphi, i = 1, \ldots, n$	φ holds, as long as one of the interacting features is absent.

Implicit definition of feature composition

$(F_1 \oplus F_2) \equiv F$ iff $\forall S(S \oplus F \equiv (S \oplus F_1) \oplus F_2)$ where $S_1 \equiv S_2 \rightleftharpoons \forall\varphi(S_1 \models \varphi \Leftrightarrow S_2 \models \varphi)$.

Furthermore, we assume that the chosen temporal logic allows the representation of $S \models \varphi$ in the form $\models [\![S]\!] \Rightarrow \varphi$, where $[\![S]\!]$ stands for a logical formula which defines the class of the logic's models which represent behaviours of S.

In this paper we propose a representation for $\oplus$. We assume that a *feature ready* form $S_n \ldots S_1 B$ of $[\![S]\!]$ can be obtained in which the immutable base of S is denoted by formula B and the parts of S which can be affected by the integration of features are denoted by substitutions $S_1, \ldots, S_n$, S_i denoting parts of S which are mutable relative to the ith "stub" $S_{i-1} \ldots S_1 B$. We assume that a feature F is described as a sequence $F_1, \ldots, F_n$ of substitutions which represent the additions made upon its integration at the various levels of mutability in S, and the way it changes the roles of the default mutable parts of S, so that $[\![S \oplus F]\!]$ can be put down as $S_n F_n \ldots S_1 F_1 B$. Here n and the particular predicate letters subject to instantiation can vary, depending on the kind of systems and features in question.

Our approach was inspired by Samborski's *stack service model* [12] and builds on the semantics of the *feature construct* for SMV [10], given in [9]. We show that several description formalisms for features known from the literature can be fit into the formal framework we propose. Among these formalisms are Samborski's model [12] and the *feature construct* for SMV [10].

We choose the extension of the Duration Calculus (DC, [15, 6]) by iteration [1], a quantifier which binds state variables [8] and one which binds temporal variables with finite variability [5], as the logic for our examples in this paper. We include a concise definition of DC here only. The properties of DC that motivate our choice become manifest by the way we use it to describe systems' behaviour. We believe that other choices are compatible with our approach too.

1 Preliminaries on the Duration Calculus and Substitution of Predicate Letters

DC is a linear time first order interval-based temporal logic. It has one normal binary modality known as *chop*. A comprehensive survey on DC can be found in [6]. Extensions and variants of DC have been proposed and studied in a number of works, among which are [1, 14, 8, 13]. Since we use some of these extending constructs, and for the sake of self-containedness, we include a brief formal definition of DC as it appears in this paper.

Languages Along with the customary first order logic symbols, DC vocabularies include *state variables*. State variables P form *state expressions* S, which have the syntax:

$S ::= \mathbf{0} \mid P \mid (S \Rightarrow S)$

State expressions occur in formulas as part of *duration terms* $\int S$. The syntax of DC *terms* t and *formulas* φ extends that of first order logic by duration terms and formulas built using the modalities *chop* and *iteration*, denoted here by $(.;.)$ and $(.)^*$, respectively:

$t ::= c \mid x \mid \int S \mid f(t, \ldots, t)$

$\varphi ::= \bot \mid R(t, \ldots, t) \mid (\varphi \Rightarrow \varphi) \mid (\varphi; \varphi) \mid \varphi^* \mid \exists x \varphi$

Constant, function and predicate symbols can be either *rigid* or *flexible* in DC. (The interpretations of rigid symbols are required not to depend on the reference interval.) Individual variables are rigid. State variables are flexible. The symbols 0, $+$, $=$ and $\leq$ with their customary roles are mandatory in DC vocabularies. In the BNF for formulas, x stands for either an individual variable, or a state variable, of a flexible constant. Flexible constants are also called *temporal variables* in DC.

Semantics The model of time in DC is the linearly ordered group of the reals, which is also the fixed domain of individuals of DC. Other models of time have been studied too. Domains are constant in DC in general. The set of the possible worlds in models for DC in the Kripke sense is $\{[\tau_1, \tau_2] : \tau_1, \tau_2 \in \mathbf{R}, \tau_1 \leq \tau_2\}$. We denote this set by $\mathbf{I}$. A DC *interpretation* I *of a* DC *language* $\mathbf{L}$ is a function on $\mathbf{L}$'s vocabulary. The types of the values of I for symbols of the various kinds are as follows:

$I(x), I(c) \in \mathbf{R}$	for individual variables x and rigid constants c
$I(c) : \mathbf{I} \to \mathbf{R}$	for flexible c
$I(f) : \mathbf{R}^n \to \mathbf{R}$, $I(R) : \mathbf{R}^n \to \{0, 1\}$	for n-ary rigid f, R
$I(f) : \mathbf{I} \times \mathbf{R}^n \to \mathbf{R}$, $I(R) : \mathbf{R}^n \to \{0, 1\}$	for n-ary flexible f, R
$I(P) : \mathbf{R} \to \{0, 1\}$	for state variables P

$I(0)$, $I(+)$, $I(\leq)$ and $I(=)$ are required to be the corresponding components of $\langle \mathbf{R}, 0, +, \leq \rangle$ and equality on $\mathbf{R}$, respectively.

Interpretations $I(P)$ of state variables P are required to have the *finite variability property* which means that for any two $\tau_1, \tau_2 \in \mathbf{R}$, the set $\{\tau : I(P)(\tau) = 0 \text{ and } \tau_1 \leq \tau \leq \tau_2\}$ is required to be either empty, or a finite union of intervals. This requirement corresponds to the assumption that observable states change only finitely often in bounded intervals of time.

Given an interpretation I, the value $I_\tau(S)$ of state expression S at time $\tau \in \mathbf{R}$, and the value $I_\sigma(t)$ of a term t at interval $\sigma \in \mathbf{I}$ are defined by the clauses:

$$
\begin{array}{lcl}
I_\tau(\mathbf{0}) & = & 0 \\
I_\tau(P) & = & I(P)(\tau) \\
I_\tau(S_1 \Rightarrow S_2) & = & \max\{1 - I_\tau(S_1), I_\tau(S_2)\}
\end{array}
$$

$$
\begin{array}{lcl}
I_\sigma(c) & = & I(c)(\sigma) \\
I_\sigma(x) & = & I(x) \\
I_\sigma(\int S) & = & \int\limits_{\min \sigma}^{\max \sigma} I_\tau(S) d\tau \\
I_\sigma(f(t_1, \ldots, t_n)) & = & I(f)(I_\sigma(t_1), \ldots, I_\sigma(t_n)) \text{ for rigid } f \\
I_\sigma(f(t_1, \ldots, t_n)) & = & I(f)(\sigma, I_\sigma(t_1), \ldots, I_\sigma(t_n)) \text{ for flexible } f
\end{array}
$$

The modelling relation $\models$ is defined on interpretations I of a given language $\mathbf{L}$, intervals $\sigma \in \mathbf{I}$ and formulas φ from $\mathbf{L}$ by the clauses:

$I, \sigma \not\models \bot$
$I, \sigma \models R(t_1, \ldots, t_n)$ iff $I(R)(\sigma, I_\sigma(t_1), \ldots, I_\sigma(t_n)) = 1$;
$I, \sigma \models \varphi \Rightarrow \psi$ iff either $I, \sigma \models \psi$ or $I, \sigma \not\models \varphi$;
$I, \sigma \models (\varphi; \psi)$ iff $I, \sigma_1 \models \varphi$ and $I, \sigma_2 \models \psi$ for some $\sigma_1, \sigma_2 \in \mathbf{I}$ such that $\sigma = \sigma_1 \cup \sigma_2$ and $\min \sigma_2 = \max \sigma_1$;
$I, \sigma \models \varphi^*$ iff there exists an ascending sequence $\tau_0, \ldots, \tau_n$ such that $\tau_0 = \min \sigma$, $\tau_n = \max \sigma$, and $I, [\tau_{i-1}, \tau_i] \models \varphi$, $i = 1, \ldots, n$;
$I, \sigma \models \exists x \varphi$ iff $J, \sigma \models \varphi$ for some J which is a x-variant of I.

Abbreviations The symbols $\top$, $\neg$, $\vee$, $\wedge$, $\Leftrightarrow$, $\forall$, $\neq$, $\geq$, $<$ and $>$ are used to abbreviate formulas and terms in the usual way. Infix notation is used wherever $+$, $=$ and $\leq$ occur. The following abbreviations are (more) DC-specific:

$$\mathbf{1} \rightleftharpoons \mathbf{0} \Rightarrow \mathbf{0} \quad \ell \rightleftharpoons \textstyle\int \mathbf{1} \quad \lceil S \rceil \rightleftharpoons \ell \neq 0 \wedge \textstyle\int S = \ell \quad \Diamond\varphi \rightleftharpoons ((\top; \varphi); \top) \quad \Box\varphi \rightleftharpoons \neg\Diamond\neg\varphi$$

When omitting parentheses, we assume that $(.;.)$ has the *smallest* binding strength. We never omit the parentheses of $(.;.)$ itself. We also write $(\varphi; \psi; \chi)$ instead of $((\varphi; \psi); \chi)$, etc.

Substitution of predicate letters by predicates This is our main tool here. In this paper *variables* means individual, state or temporal variables. Let φ be a formula and $x_1, \ldots, x_n$ be free variables of φ. Let $\lambda x_1 \ldots x_n.\varphi$ be the predicate on $x_1, \ldots, x_n$ which φ defines. We do not require $FV(\varphi) \subseteq \{x_1, \ldots, x_n\}$, in order to enable the definition of parameterised families of predicates by single φs. Given an n-ary predicate letter P and a formula ψ, the *substitution* $[\lambda x_1 \ldots x_n.\varphi/P]\psi$ *of* P *by* $\lambda x_1 \ldots x_n.\varphi$ is defined by the clauses:

$\theta\bot \rightleftharpoons \bot$
$\theta R(t_1, \ldots, t_m) \rightleftharpoons R(t_1, \ldots, t_m)$, if $R \neq P$
$\theta P(t_1, \ldots, t_n) \rightleftharpoons [t_1/x_1, \ldots, t_n/x_n]\varphi$
$\theta(\psi_1 \Rightarrow \psi_2) \rightleftharpoons \theta\psi_1 \Rightarrow \theta\psi_2$
$\theta(\psi_1; \psi_2) \rightleftharpoons (\theta\psi_1; \theta\psi_2)$
$\theta(\varphi^*) \rightleftharpoons (\theta\varphi)^*$
$\theta\exists x\psi \rightleftharpoons \exists y \theta[y/x]\psi$ where $y \notin FV(\varphi) \setminus \{x_1, \ldots, x_n\}$

Simultaneous substitution $[\lambda x_1 \ldots x_{n_i}.\varphi_i/P_i : i \in I]$ of several predicate letters is defined similarly.

2 Hardware features: an introductory example

Let us show how our approach works when the integration of a feature amounts to connecting a piece of circuitry to a system, which itself is a circuit. Let the observable signals of our basic system be $x_1, \ldots, x_n$. Let $x_1, \ldots, x_n$ be also the names of the state variables which stand for these signals in the formula S describing the system. That is, an interpretation I for the signals $x_1, \ldots, x_n$ represents a behaviour of the considered basic system at interval σ iff $I, \sigma \models [\![S]\!]$. Naturally, $FV([\![S]\!]) \subseteq \{x_1, \ldots, x_n\}$. There are extensive studies on the description of hardware and especially digital circuits by DC in this way. For instance, [11] proposes a set of derived DC constructs called *implementables* for the description of the basic temporal and causal relations between input and output signals of the simplest digital devices.

Let φ be a formula which represents the circuitry that comes with the feature F in a similar way. Let $FV(\varphi) = \{y_1, \ldots, y_m\}$. To describe the integration of the feature completely, we need to list the connections between the signals $x_1, \ldots, x_n$ of the basic system and the signals

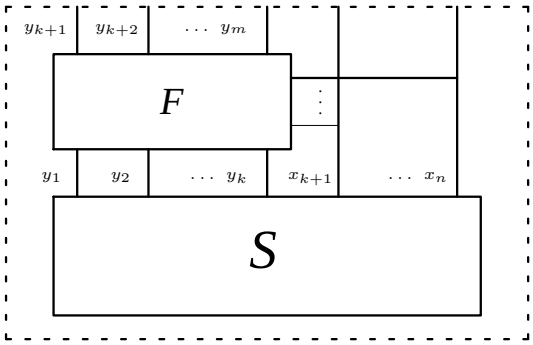

Figure 1: Feature F is integrated into system S by adding a circuit. S and $S \oplus F$, which is represented by the dashed rectangle, have the same observable signals.

of the new circuit $y_1, \ldots, y_m$. Some of these signals get identified, that is, connected by conductors, others remain accessible to the environment. Let the signals $x_1, \ldots, x_k$ of S be connected to the signals $y_1, \ldots, y_k$ of the feature. Let the system obtained after the integration interact with its environment through the remaining signals $x_{k+1}, \ldots, x_n, y_{k+1}, \ldots, y_m$ of its now two parts. Then its behaviour can be described by the formula

$$[\![S \oplus F]\!] = [x_1/y_{k+1}, \ldots, x_k/y_{2k}]\exists y_1 \ldots \exists y_k(\varphi \wedge [y_1/x_1, \ldots, y_k/x_k][\![S]\!]) \tag{1}$$

We assume that $m = 2k$, that is, the number of signals that the feature contributes to the system is equal to the number of signals it hides from the environment, for the sake of simplicity. Then S can be written as

$$\underbrace{[\lambda x_1 \ldots x_n.[\![S]\!]/P]}_{S_1}\underbrace{P(x_1, \ldots, x_n)}_{B} \tag{2}$$

and $[\![S \oplus F]\!]$ can be written as

$$S_1[\lambda y_{k+1} \ldots y_m x_{k+1} \ldots x_n.\exists y_1 \ldots \exists y_k(\varphi \wedge P(y_1, \ldots, y_k, x_{k+1}, \ldots, x_n))/P]B \tag{3}$$

respectively, where P is an n-ary predicate letter. Since

$$[\lambda x_1 \ldots x_n.[\![S]\!]/P]P(y_1, \ldots, y_k, x_{k+1}, \ldots, x_n) \text{ is } [y_1/x_1, \ldots, y_k/x_k][\![S]\!],$$

the formulas (1) and (3) are equivalent. Besides, (3), which describes $S \oplus F$, is obtained by inserting a substitution between S_1 and B in the description (2) for S. The inserted substitution carries all the information relevant to the integration of the feature F. The formula $P(x_1, \ldots, x_n)$ on the right in both (2) and (3) determines the interface of the system with the environment, that is the list of its signals. The substitution $[\lambda x_1 \ldots x_n.[\![S]\!]/P]$ determines both the implementation of S as before the introduction of F and the behaviour of the circuit found in S upon the integration of F as part of $S \oplus F$.

Some observations can be made on the way the integration of a feature was described above.

The description $[\![S]\!]$ of S was rewritten into the form (2), as the instantiation of the system's interface B, by the system's actual behaviour S_1. This roughly corresponds to dismantling the system and preparing it for an upgrade. Upon dismantling the system, the identity between the implementation of its actual behaviour "inside" and its interface "outside" is lost,

and the place holder P which is now explicitly available for substitution, defines the possible ways of revising this identity. The revision comes in the form of the intermediate substitution F which describes the feature to be integrated.

Here, F can fully control the extent to which the basic system's circuit affects the featured system's behaviour. For instance, F can choose to fully reinterpret the signals $x_1, \ldots, x_n$ of S; it can also reinterpret the signals $x_1, \ldots, x_k$ only, as we have done above.

The place holder P has at most one occurrence in the definition of the predicate on the left hand side of $/$ in F. This occurrence is positive and in the scope of existential quantifiers only. This is related to the feature being an addition of circuitry. Multiple occurrences of P would mean that whatever circuitry implements $\lambda x_1 \ldots x_n.[\![S]\!]$, should be multiplied upon the integration of F. This is unrealistic for hardware. Similarly, a negative occurrence or an occurrence in the scope of a universal quantifier may not correspond to any conceivable reassembly of the circuitry of S. In the next section we deal with software features, where this needs not be the case.

The description of the circuits of S and F as predicates on their signals is compatible with the approach to use *modules* and module *instances* as known from, e.g., SMV.

3 Software features

The huge variety of meanings that *executing programs* has in the various programming languages makes it very hard to search for universal and practically valuable formalisations to the integration of software features. In this section we introduce a simple imperative real time concurrent programming language similar to that in [3], in order to illustrate how software features' behaviour can be described by substitution. Programs in it are fixed sets of interleaving processes with shared variables. We assume that they interact with their environments by reading and writing signals like variables. We define the semantics of the language by a translation of programs into DC formulas.

3.1 A language for concurrent processes with shared variables

Programs $\mathbf{P}$ in our programming language are parallel compositions of sequential processes of the form

$$\mathbf{P} = P_1 \| \ldots \| P_n \tag{4}$$

which all have access to the same set of variables and input and output signals. We use x, y, $\ldots$ to denote both variables and signals. The only difference is that signals may not occur on the wrong side of assignment statements. In the BNF for processes below e and c stand for arbitrary and boolean expressions respectively:

$P ::= \texttt{skip} \mid x := e \mid \texttt{delay}\ e \mid \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ P \mid (P; P) \mid \texttt{while}\ c\ \texttt{do}\ P$

We denote the set of the variables occurring on the left side of assignment statements in process P by $Write(P)$. The DC language we use to describe the behaviour of programs contains a pair of temporal variables x and x' for every syntactically correct program variable x, the state variables $active_i$, and the flexible 0-ary predicate letters $[\![P]\!]_i$, $i < \omega$, for every syntactically correct process P.

The DC temporal variables x and x' denote the values of the program variable x in the beginning and in the end of a reference interval, respectively, $active_i$ holds at time τ iff P_i is active at time τ, and $[\![P]\!]_i$ hold at interval σ iff σ represents a complete run of a subprocess

P of the process P_i in (4). We consider only terminating behaviour here. Nonterminating behaviour can be handled similarly, using the extension of DC by *infinite intervals* [13].

For the rest of this section we fix a program $\mathbf{P}$ and $\models_{\mathbf{P}}$ to denote validity in the DC *theory of* $\mathbf{P}$. Obviously, for all program variables x

$$\models_{\mathbf{P}} \forall u \forall v((x' = u; x = v) \Rightarrow u = v) \tag{5}$$

Consider the abbreviations:

$\mathsf{K}(X) \rightleftharpoons \bigwedge_{x \in X} \Box x' = x$, where X stands for a set of variables;

$A_i \rightleftharpoons \lceil active_i \rceil$;

$Sleep_i \rightleftharpoons \lceil \neg active_i \rceil \vee \ell = 0$.

$\mathsf{K}(X)$ describes that the variables X preserve their values within the reference interval. A_i describes that process P_i is active and $Sleep_i$ describes that P_i is inactive throughout the reference interval. Obviously, $\models_{\mathbf{P}} A_i \Rightarrow \Box(A_i \vee \ell = 0)$ for all i and $\models_{\mathbf{P}} \neg(A_i \wedge A_j)$ if $i \neq j$. A variable x can be updated only *within* an interval where some process is active:

$$\models_{\mathbf{P}} \Box\left(\neg \mathsf{K}(\{x\}) \Rightarrow \bigvee_{x \in Write(P_i)} \Diamond A_i\right)$$

Let t_e denote the time needed to evaluate expression e. Then the predicate letters $[\![P]\!]_i$ validate the formulas:

$\models_{\mathbf{P}} [\![\texttt{skip}]\!]_i \Leftrightarrow Sleep_i$

$\models_{\mathbf{P}} [\![x := e]\!]_i \Leftrightarrow (\ell = t_e \wedge x' = e \wedge \mathsf{K}(Write(P_i) \setminus \{x\}) \wedge A_i; Sleep_i)$

$\models_{\mathbf{P}} [\![\texttt{delay}\ e]\!]_i \Leftrightarrow \ell = \max(t_e, e) \wedge (A_i \wedge \mathsf{K}(Write(P_i)); Sleep_i)$

$\models_{\mathbf{P}} [\![\texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q]\!]_i \Leftrightarrow \left(\begin{array}{l}(\ell = t_c \wedge c \wedge A_i \wedge \mathsf{K}(Write(P_i)); Sleep_i; [\![P]\!]_i) \vee \\ (\ell = t_c \wedge \neg c \wedge A_i \wedge \mathsf{K}(Write(P_i)); Sleep_i; [\![Q]\!]_i)\end{array}\right)$

$\models_{\mathbf{P}} [\![(P; Q)]\!]_i \Leftrightarrow ([\![P]\!]_i; [\![Q]\!]_i)$

$\models_{\mathbf{P}} [\![\texttt{while}\ c\ \texttt{do}\ P]\!]_i \Leftrightarrow [\![\texttt{if}\ c\ \texttt{then}\ (P; \texttt{while}\ c\ \texttt{do}\ P))\ \texttt{else}\ \texttt{skip}]\!]_i$

Under these assumptions, an interpretation I and an interval σ represent a complete run of $\mathbf{P}$ iff

$$I, \sigma \models_{\mathbf{P}} \bigwedge_{i=1}^{n} [\![P_i]\!]_i \tag{6}$$

3.2 Adding features which are processes

Designing a software feature as a process to be run concurrently with the rest of the basic system's processes is intuitive, because the design can be started from an informal account of what conditions on the behaviour of a system should trigger action on behalf of a feature and how the feature should affect the working of the system. Besides, a process can easily be designed to monitor the behaviour of the processes it shares variables with.

Let F be a feature of this kind. Let P_f be the process to be added upon the integration of F and (4) stand for the basic system. Then the result $\mathbf{P} \oplus F$ of integration F into $\mathbf{P}$ can be represented as $\mathbf{P} \| P_f$. This suggests the following form of the basic system, which is feature-ready for features which are additional concurrently run processes:

$$\mathbf{P} = [\texttt{skip}/A]P_1 \| \ldots P_n \| A$$

Now the substitution $[P_f \| A/A]$ can be used to describe the integration of F:

$$\mathbf{P} \| P_f = [\mathtt{skip}/A][P_f \| A/A]P_1 \| \ldots \| P_n \| A$$

Since the behaviour of $\mathbf{P}$ is described by means of a simple conjunction in (6), a similarly shaped feature-ready form of $[\![\mathbf{P}]\!]$ can be obtained, and (6) can be rewritten in the form:

$$I, \sigma \models_{\mathbf{P}} [Sleep_a/A] \left(\bigwedge_{i=1}^{n} [\![P_i]\!]_i \right) \wedge A \tag{7}$$

where $a \notin \{1, \ldots, n\}$, and then I, σ would represent a complete run of $\mathbf{P} \oplus F$ iff

$$I, \sigma \models_{\mathbf{P}} [Sleep_a/A][[\![P_f]\!]_f \wedge A/A] \left(\bigwedge_{i=1}^{n} [\![P_i]\!]_i \right) \wedge A \tag{8}$$

where $f \notin \{1, \ldots, n, a\}$.

3.3 Adding features which are procedures

Software features can also be implemented as modifications of subroutines such as drivers and operating system functions which are to be invoked through standard entry points. One way to demonstrate this in our programming language is to introduce named processes and allow occurrences of their names to stand for invocations of these subprocesses as procedures. Yet, in order to show how substitution can be used to describe the revision of such procedures upon the integration of a feature, we only extend the BNF for processes to allow place holders A to occur where subprocesses subject to revision is to be put:

$P ::= \mathtt{skip} \mid \ldots \mid \mathtt{while}\ c\ \mathtt{do}\ P \mid A$

Now a feature-ready program can be written in the form

$$\mathbf{P} = [D_1/A_1, \ldots, D_k/A_k]P_1(A_1, \ldots, A_k) \| \ldots \| P_n(A_1, \ldots, A_k) \tag{9}$$

where the parameter lists $(A_1, \ldots, A_k)$ indicate that the processes $P_1, \ldots, P_n$ may have been written with occurrences of the some of place holders $A_1, \ldots, A_k$, and $D_1, \ldots, D_k$ are the default implementations of the processes named $A_1, \ldots, A_k$, which are part of the basic system. Various kinds of substitutions can be inserted in (9) to change the working of $D_1, \ldots, D_k$ in various ways:

$[P_{f,i}/A_i]$	The default procedure D_i gets replaced by one supplied by the feature. Further feature integration may be unable to change this again, because $P_{f,i}$ contains no place holders.
$[(P_{f,i}; A_i)/A_i]$	D_i gets *chained*, that is, programmed to be run after some new code $P_{f,i}$. This is typical of exception handlers.
$\left[\begin{array}{l}\mathtt{if}\ c \\ \quad\mathtt{then}\ P_{f,i} \\ \quad\mathtt{else}\ A_i\end{array}/A_i\right]$	The feature-supplied code replaces the default conditionally.

Just like in the case of adding processes by parallel composition, the role of place holders and the corresponding substitutions is preserved in the DC description of the behaviour of the considered systems.

3.4 The stack service model in terms of substitutions

Samborski's *stack service model* assumes that services in a distributed system are organised as a *network* of *stacks*, and process *tokens*, which represent user and system requests. A stack processes a token it receives by feeding it to its topmost service. A service may choose either to completely process a token itself, or resort to the service immediately below it in the stack by generating a token to be passed to this service. In this model feature integration is represented by the insertion of services into stacks. Different stacks in a network can have different services and have them appearing in different orders. This reflects the possibility for different users of the network to subscribe to different features, as known in the case of a telephone system, and prioritise the reaction of services in response to requests in different ways. In the stack service model, feature integration is done by inserting services in stacks.

The *state* of a stack of services is a tuple

$$\mathbf{S} = \langle n, L_u, L_s, \varepsilon, d, R_1 \cdot \ldots \cdot R_k, V \rangle$$

where n is an *identifier* for the stack in question, L_u and L_s are queues of incoming tokens originating from the stack's user and from the system, respectively, $\varepsilon \in \{0, 1\}$ denotes the state of a "door" which gives priority to user incoming tokens when open, and to system tokens when closed, d is the *coming down* token, $S = R_1 \cdot \ldots \cdot R_k$ is the *list of services* in $\mathbf{S}$, and V is the *valuation of the variables* of $\mathbf{S}$. If there is no coming down token d, δ is put in the place of d. Services R_i, $i = 1, \ldots, k$, can be regarded as functions which, given a token t, a state of the "door" ε and a valuation of the stack's variables V, return a tuple $\langle d, U, \gamma, W \rangle$, where d is a token to be passed to the next service in the stack, U is a set of tokens to be released in the network, and γ and W are the state of the "door" and the valuation of the stack's variables upon the completion of R. In case service R does not process token t, $R(t, \varepsilon, V) = \langle t, \emptyset, \varepsilon, V \rangle$.

The operational semantics of the stack service model is presented in detail in [12]. In this section we show how the feature integration in the sense of the stack service model can be described in terms of substitutions.

3.4.1 Programming stacks in the language of shared variables

A stack of services can be programmed as a process in the extension of our programming language by one process place holder A. Individual services R can be programmed in the form

$$P(R) = (C(R); \texttt{if } d \neq \delta \texttt{ then } A \texttt{ else skip}), \tag{10}$$

where $C(R)$ stands for some code which implements the relation $R(t, \varepsilon, V) = \langle d, U, \gamma, W \rangle$ by sampling and assigning ε, d and $x_1, \ldots, x_m$ appropriately, and releasing the tokens U into the network by means of, e.g., an atomic command `putToken` (e). The designated occurrence of a place holder A is the only one allowed in (10). It enables the chaining of successive services $\ldots R_i \cdot R_{i+1} \ldots$ in a stack by means of a sequence of successive substitutions

$$\ldots [[\![P(R_{i+1})]\!]_n / A][[\![P(R_i)]\!]_n / A] \ldots$$

Here n is the identifier of the stack. Let the variables of a stack with identifier n, be $x_1^n, \ldots, x_m^n$. $P(R)$ can be programmed to use the same names $x_1, \ldots, x_m$ for the variables $x_1^n, \ldots, x_m^n$ of

whatever stack R can be part of, and Id for the identifier of this stack. Let θ_n be the substitution $[n/Id, x_1^n/x_1, \ldots, x_m^n/x_m]$. Then $\theta_n P(R)$ will stand for the instance of $P(R)$ which can appear in the implementation of a stack with identifier n. We assume that the queues L_u and L_s, the door ε, the coming-down token d and an auxiliary variable t are among $x_1, \ldots, x_m$ for the sake of simplicity.

Let b' be an auxiliary process to acquire a coming-down token from the appropriate queue and transfer control to the topmost service of the stack:

(`while` $L_u = \emptyset \wedge L_s = \emptyset$ `do skip`;
`if` $\varepsilon \wedge L_u \neq \emptyset$ `then` (
$d := head(L_u); L_u := tail(L_u); \varepsilon = 0$
) `else` (
$d := head(L_s); L_u := tail(L_s)); A$
)
)

Let b'' be an auxiliary process to feed tokens to L_u and L_s from the network. Let, given the stack identifier Id, the $TokensAvailable(Id)$ indicate whether the network contains tokens bound for this stack. Let `getToken` (Id, t) assign t a Id-bound token from the network, if it contains any tokens bound for this stack. Then b'' can be programmed by means of the auxiliary boolean function $TokensAvailable$ and atomic command `getToken` as follows:

`if` $TokensAvailable(Id)$ `then` (
`getToken` (Id, t);
`if` $UserToken(t)$ `then` $L_u := L_u * t$ `else` $L_s := L_s * t$
)
`else skip`

Let s_1 be an auxiliary process to output to the network whatever token the last service in the stack produces:

`if` $d \neq \delta$ `then` (`putToken` (d); $d := \delta$) `else skip`

Now, given that $R_1 \cdot \ldots \cdot R_k$ is the list of services in the stack, the behaviour of the stack can be represented by the formula

$$\theta_n[[\![s_1]\!]_n/A][[\![P(R_k)]\!]_n/A] \ldots [[\![P(R_1)]\!]_n/A]([\![b']\!]_n^* \wedge [\![b'']\!]_{n'}^*) \,. \tag{11}$$

The occurrence of A in $P(R_i)$ guarantees that upon terminating R_i transfers control to R_{i+1} for $i = 1, \ldots, k-1$, and R_k transfers control to s_1. Control gets transferred only if R_i terminates with $d \neq \delta$. The index n' in $[\![b'']\!]_{n'}$ here can be chosen to be e.g. $N + n + 1$, where N is the greatest value that a stack identifier can take, to ensure that the two processes corresponding to each stack in a system interleave correctly.

To describe the behaviour of several stacks running concurrently in the same network, one can take the conjunction of the formulas (11) with n ranging over the identifiers of the stacks in the system and $R_1^n \cdot \ldots \cdot R_{k_n}^n$, being the list of services of stack with identifier n.

4 The SMV feature construct in terms of substitutions

In this section we show that the feature construct introduced in [10] to the input language of SMV [7] fits into our scheme of representing feature integration as the insertion of substitutions. We first show how the description of a system in SMV can be translated into an appropriate DC formula, written using substitutions of predicate symbols. The predicate symbols subject to these substitutions and the predicates which substitute them stand for the names

of the components of the modelled system which are subject to change by the integration of features and these components themselves, respectively. Finally, the integration of features as it can be described using the SMV feature construct is done by inserting appropriate substitutions in the original formula.

Assumptions about the initial SMV description and its form

For the sake of simplicity, we assume that the description of the basic, that is, feature-free, system consists of a single main module.[1] We assume that all the assignment statements in the considered main module are of one of the two forms:

$\texttt{next}(\langle variable \rangle) \ := \ \langle expression \rangle;$
$\texttt{init}(\langle variable \rangle) \ := \ \langle expression \rangle;$

and all variables occur in the left hand side of *exactly* one assignment of each of the two kinds, possibly with non-deterministic expressions on the right hand side. We assume that the expressions above are `case` expressions with disjoint guards and possibly nondeterministic unconditional subexpressions. (All SMV programs can be rewritten into this form.)

Under all the above restrictions, such an arbitrary single simple SMV module can be regarded as a set of variables X, a set of next value assignments and initial value assignments:

$$\begin{array}{ll}
\texttt{next}(x) := \texttt{case} & \texttt{init}(x) := \texttt{case} \\
\quad g_{x,0} \ : \ v_{x,0}\texttt{;} & \quad h_{x,i} \ : \ w_{x,0}\texttt{;} \\
\quad \vdots & \quad \vdots \\
\quad g_{x,N_x-1} \ : \ v_{x,N_x-1}\texttt{;} & \quad h_{x,i} \ : \ w_{x,M_x-1}\texttt{;} \\
\texttt{esac} & \texttt{esac}
\end{array}$$

Given x, let e_x and f_x stand for the expressions on the right side of the corresponding next and initial value assignments.

We deal with the `treat` and the `impose` components of the feature construct for SMV first here.

As in the previous sections, given the set X of the variables which occur in the considered system description, we assume that the vocabulary of our logical language contains the temporal variables named x and x' for every $x \in X$. We also assume that the syntax of the gs, vs, hs and ws is the same as that of the terms in the chosen logical language. Using this convention, a description of any finite initial subinterval of a behaviour of a system can be written in the form:

$$\bigwedge_{x \in X} \Big(\Big(\ell = 1 \wedge \bigwedge_{i<N_x} (g_{x,i} \Rightarrow \bigvee_{v \in v_{x,i}} x' = v) \Big)^* \wedge \Big(\bigwedge_{i<M_x} (h_{x,i} \Rightarrow \bigvee_{w \in w_{x,i}} x = w; \top) \Big) \Big) \qquad (12)$$

Here, as above, we assume that the temporal variables x and x' from our logical vocabulary denote the initial and the final values of the corresponding SMV variable $x \in X$ in every reference interval, and therefore satisfy the axiom (5).

[1] Of course, what comes below is hardly like what users of SMV have these days. Arguably, it only has comparable potential expressivity.

Making the description feature-ready

In order to enable the alteration of a system's description by means of substitution, we need to have symbols to subject to substitution at the places which we intend to allow the integration of features to affect. (This is so, because substitution is simplest to define if its target is an atomic entity.) The `impose` and `treat` statements in the SMV feature construct prescribe the alteration of expressions to be assigned to variables and the alteration of the meaning of variable occurrences in such expressions. This means that we need to be able to substitute these expressions and the occurrences of variables in them. Variable occurrences are represented by atomic entities in (12), but the possibly ambiguous right hand sides of SMV assignments are represented as flexible predicates on the variables x and x', which, given $x \in X$, stand for $\mathtt{init}(x)$ and $\mathtt{next}(x)$ in (12). To introduce symbols in their places, we rewrite (12) in the form:

$$\left[\begin{array}{lll} \lambda u. \bigwedge\limits_{i<N_x} (g_{x,i} \Rightarrow \bigvee\limits_{v \in v_{x,i}} u = v) & / & E_x, \\ \lambda u. \bigwedge\limits_{i<M_x} (h_{x,i} \Rightarrow \bigvee\limits_{w \in w_{x,i}} u = w) & / & F_x \end{array} : x \in X \right] \underbrace{\bigwedge_{x \in X} ((\ell = 1 \wedge E_x(x'))^* \wedge (F_x(x); \top))}_{B} \tag{13}$$

Clearly, (12) and (13) evaluate to the same formula. Using the predicate letters E_x and F_x as place holders enables the presentation of practically arbitrary restrictions on the initial and subsequent values of the variables $x \in X$. We need this, because of the possibility to have nondeterministic assignments. Similarly, although variables' occurrences on the right hand side of assignment are atomic, it is not sufficient to use them immediately as place holders to be substituted by terms, because SMV expressions which occur in `treat` statements can be nondeterministic too, and therefore produce more than a single term in the chosen logical form of description. That is why we further partition (12). To do this, we assume that the temporal variables $\tilde{x}$, $x \in X$, are fresh wherever they occur, and we introduce the fresh unary predicate letters V_x, $x \in X$. Informally, $V_x(u)$ denotes that u equals the current value of the SMV variable x. We denote the set of SMV variables which occur in a given SMV expression e by $Var(e)$. Given that $v_1, \ldots, v_n$ is a finite set of SMV variables, we abbreviate the quantifier prefix $\exists \tilde{v_1} \ldots \exists \tilde{v_n}$ by $\exists_{v \in \{v_1, \ldots, v_n\}} \tilde{v}$. Using this notation, we replace (13) by

$$\underbrace{[\lambda u.u = x / V_x : x \in X]}_{S_2} \underbrace{S_1' S_1''}_{S_1} B \tag{14}$$

where S_1' and S_1'' denote

$$[\lambda u. \exists_{y \in Var(e_x)} \tilde{y} (\bigwedge_{y \in Var(e_x)} V_y(\tilde{y}) \wedge [\tilde{y}/y : y \in Var(e_x)] \bigwedge_{i<N_x} (g_{x,i} \Rightarrow \bigvee_{v \in v_{x,i}} u = v)) / E_x, : x \in X]$$

and

$$[\lambda u. \exists_{y \in Var(f_x)} \tilde{y} (\bigwedge_{y \in Var(f_x)} V_y(\tilde{y}) \wedge [\tilde{y}/y : y \in Var(f_x)] \bigwedge_{i<M_x} (h_{x,i} \Rightarrow \bigvee_{w \in w_{x,i}} u = w)) / F_x : x \in X]$$

respectively The advantage of this apparently longer form of description is that now the effect of integrating features with `treat` statements can be represented by substitutions on V_x, to be inserted between S_2 and S_1. Similarly, integrating features with `impose` statements can be represented by substitutions on E_x and F_x, to be inserted between S_1 and B. The precise form of these substitutions is explained below. Clearly, (13) is equivalent to (14). In the sequel we briefly denote (14) by $S_2 S_1 B$.

Describing the integration of a feature

Consider the SMV feature which, for the sake of simplicity, consists of the single `impose` statement targetting some variable $y \in X$

`if` c_i `then impose next(`y`) :=` e_i;

and the single `treat` statement targetting some variable $z \in X$

`if` c_t `then treat` z = e_t;

Let e_i and e_t be of the same form as e_x and f_x, $x \in X$, above. Predicates like the ones in S_1' and S_1'' for e_x and f_x, $x \in X$, can be written to represent e_i and e_t too. Let us denote these unary predicates by $[\![e_i]\!]$ and $[\![e_t]\!]$ respectively, and assume, just as in the case of e_x and f_x, that they are defined in terms of $Var(e_i)$ and $Var(e_t)$, respectively. Then, given the description S_2S_1B of the feature-free system considered above, the result of integrating this feature into this system can be described by

$$S_2(S_tS_1)|_{\{E_x,F_x:x\in X\}}S_iB \tag{15}$$

where $S|_A$ denotes the restriction of a substitution S to a set of symbols A, and

$$S_t = [\lambda u.\exists_{y\in Var(e_t,c_t)}\tilde{y}(\bigwedge_{y\in Var(e_t,c_t)} V_y(\tilde{y}) \wedge [\tilde{y}/y : y \in Var(e_t,c_t)] \begin{pmatrix} ([\![e_t]\!](u) \wedge c_t)\vee \\ (V_z(u) \wedge \neg c_t) \end{pmatrix})/V_z]$$

$$S_i = [\lambda u.\exists_{y\in Var(e_i,c_i)}\tilde{y}(\bigwedge_{y\in Var(e_i,c_i)} V_y(\tilde{y}) \wedge [\tilde{y}/y : y \in Var(e_i,c_i)] \begin{pmatrix} ([\![e_i]\!](u) \wedge c_i)\vee \\ (E_y(u) \wedge \neg c_i) \end{pmatrix})/E_y]$$

Integrating more than one feature

The integration of complex features can be regarded as the integration of a sequence of features of the simple form studied above. In case several features get integrated into a system in a sequence, the ones which get integrated later affect the ones which get integrated earlier, but not the other way around. The straightforward way to express this in terms of substitutions is as follows.

Let $F_1, \ldots, F_r$ be features, each consisting of either a single `treat` statement or a single `impose` statement. Let $S_{t,k}$ be the substitution which corresponds to the `treat` statement of the feature F_k in the way introduced above, in case F_k has a `treat` statement, or be the vacuous (identity) substitution otherwise, $k = 1, \ldots, r$. Let, similarly, $S_{i,k}$ represent the `impose` statement of F_k, if F_k has one. Then the subsequent integration of $F_1, \ldots, F_r$ into the system described by S_2S_1B can be described by the formula

$$S_2S_{t,r}\ldots S_{t,1}S_1S_{i,1}\ldots S_{i,r}B$$

A simple way to motivate this approach is to notice that the composition of substitutions $S_tS_1S_i$ in the description (15) of a featured system has the role played by S_1 alone in the description (14) of the corresponding basic system. This means that subsequent acts of integrating features $F_k, \ldots, F_r$, should correspond to inserting substitutions $S_{i,k}, \ldots, S_{i,r}$, which correspond to `impose` statements, and substitutions $S_{t,k}, \ldots, S_{t,r}$, which correspond to `treat` statements, on the right and on the left hand sides of $S_{t,k-1}\ldots S_{t,1}S_1S_{i,1}\ldots S_{i,k-1}$, $\ldots$, $S_{t,r}\ldots S_{t,1}S_1S_{i,1}\ldots S_{i,r}$, respectively. Yet this approach is incorrect, because the composite substitution which is obtained this way prescribes that *all* the `treat` statements affect

the meaning of *all* the `impose` statements, disregarding the order of their integration. For instance, according to (14) the `treat` statement of the considered feature affects its `impose`, regardless to the intention of the feature's author, which might have been to exercise the effect of the `treat` statement on the basic system's description only.

The cause for this paradox is that a composition of substitutions of the kind of $S_tS_1S_i$ from (14) can be safely regarded as equivalent to a single substitution of the kind of S_1 only when applied to a formula like B, because the symbols E_x, $x \in X$, which are subject to substitution in B are in the domain of both S_1 and $S_tS_1S_i$. Unlike S_1, $S_tS_1S_i$ substitutes symbols from among V_x, $x \in X$ too. That is why, in order to obtain a version of $S_tS_1S_i$ which is strictly of the same form as S_1, we need to take the restriction of $S_tS_1S_i$ to the set of symbols $\{E_x : x \in X\}$ instead of the entire $S_tS_1S_i$. That is why we put the result (14) of integrating a single feature to a system in the form

$$S_2(S_tS_1)|_{\{E_x,F_x:x\in X\}}S_iB\ .$$

Now it can be safely assumed that $(S_tS_1)|_{\{E_x,F_x:x\in X\}}S_i$ plays the role of S_1 in the description of the result of integrating the considered feature into the considered system. Consequently, the integration of a sequence of single-statement features $F_1, \ldots, F_r$ can be described by substitutions in the form:

$$S_2(S_{t,r}\ldots(S_{t,1}S_1)|_{\{E_x,F_x:x\in X\}}S_{i,1}\ldots)|_{\{E_x,F_x:x\in X\}}\ldots S_{i,r}B$$

5 Verification of feature-ready descriptions and features

A Hoare triple $\{P\}\texttt{code}\{Q\}$ can be written in DC as

$$P \wedge [\![\texttt{code}]\!] \Rightarrow [\overline{x'}/\overline{x}]Q\ ,$$

where $\overline{x}$ and $\overline{x'}$ represent the vectors of the initial and the final values of program variables like above. That is why standard verification methods apply straighforwardly to the verification problems which appear in this paper. Here we only briefly mention some opportunities for verification which appear due to our choice of representation for systems and features.

Refinement of feature descriptions

Let the feature F_i be described by the substitutions $F_{i,n}, \ldots, F_{i,1}$ and let $S_nF_{i,n}\ldots S_kF_{i,k} = [\lambda x_1\ldots x_{n_i}.\varphi_j^{S\oplus F_i}/A_j : j = 1,\ldots,m]$, $i = 1,2$. Then, as long as all the occurrences of $A_1, \ldots, A_m$ in $S_{k-1}\ldots S_1B$ are positive,

$$\models S_nF_{1,n}\ldots S_kF_{1,k}S_{k-1}\ldots S_1B \Rightarrow \alpha$$

and

$$\models \varphi_j^{S\oplus F_2} \Rightarrow \varphi_j^{S\oplus F_1},\ j = 1,\ldots,m, \tag{16}$$

imply $\models S_nF_{2,n}\ldots S_kF_{2,k}S_{k-1}\ldots S_1B \Rightarrow \alpha$. The condition (16) can be used to define a relation of refinement between features of similar type.

Let us assume that no place holder occurs in more than one formula in a feature-ready description $S_n\ldots S_1B$. This restriction entails a one-to-one correspondence between the place

holders and the mutable parts of the considered system and allows us to associate requirements on mutable parts with place holders. Assume that all the place holders in the considered system have positive occurrences only. Let the place holder A occur in the substitution S_k. Let S_{k+1} be $[\ldots, \lambda x_1 \ldots x_m.\varphi_A/A, \ldots]$. Then a requirement ρ_A can be imposed on the mutable part instantiating A by putting

$$\models S_n \ldots S_{k+2}\varphi_A \Rightarrow \rho_A \,. \tag{17}$$

It is natural to assume that a mutable part of the system will satisfy its requirement, provided that its subparts satisfy their respective requirements. This means that (17) can be replaced by:

$$\models [\lambda x_1 \ldots x_m.\rho_C/C : C \in \mathbf{A}]\varphi_A \Rightarrow \rho_A \,. \tag{18}$$

where $\mathbf{A}$ stands for the set of all the place holders involved.

6 Integrating variables

Along with other things, integration of features can bring additional variables. In fact, the SMV feature construct enables the addition of variables in its full form, but we only focus on this feature construct's more specific elements in Section 4. Adding variables that are only handled individually requires no special care upon the integration of a feature. However, it is necessary to be able to vary the scope of operations which are to affect the values of collections of variables that are to be handled similarly, so that variables that features can contribute be included in the appropriate collections. Using $x = e$ to denote that variable x evaluates to e at a certain time is convenient as long as only fixed sets of variables are involved in every description. Revising collections of variables upon feature integration can be achieved, if equality $=$ is replaced by a flexible binary predicate M (for *M*emory) with its first place being of an also newly introduced sort VN for names for variables (or memory locations), and the second place being of the already known sort of variables' values.

Now unary predicates can be introduced to designate the collections of variables in question. For example, a feature-ready form of the property

$$M(v7, z) \Rightarrow M(v8, z) \wedge \ldots \wedge M(v15, z) \tag{19}$$

that enables the members of the conjunction to be changed is

$$\underbrace{[(\lambda v : VN).v = v8 \vee \ldots \vee v = v15/H]}_{S_1}\underbrace{(\forall v : VN)(M(v7, z) \wedge H(v) \Rightarrow M(v, z))}_{B}$$

Substitutions that revise the definitions of such unary predicates in feature descriptions can be used to define revisions of the scope of the respective operations upon feature integration. For example

$$S_1[(\lambda v : VN).H(v) \vee v = v16 \vee \ldots \vee v = v31/H]B$$

denotes that the value z of $v7$ is the same as that of 16 *more* variables, as compared to (19), where only 9 variables are said to have the same value.

Concluding remarks

We have shown that several formalisations for feature integration can be fit in the scheme of decomposing descriptions and recomposing them with the features included as substitutions of predicate letter place holders by (parameterised) predicates.

The feature integration construct for SMV [10] is the most complex case for the analysis we present, because of the diversity of ways in which substitution appears to implement the integration of a composite feature. Besides, the feature construct for SMV is the one where substitution appears in the most explicit form of all. Chaining of substitutions appears in the most general form in the case of the stack service model [12].

In our attempt to put all the cases in a unified framework we have found that there are readily available formal methods, such as DC, which have been developed for more general purposes, and can be part of our approach to features. This paper has been written with DC as the system of logic to illustrate the form of analysis we have pursued, because of the very small added cost on description in DC in the cases considered. However, we believe that our approach can be similarly successful with a variety of logical systems, if not for the conciseness of presentation, at least when description stages can be assisted by a tool, and with the benefit of better mechanizability of the verification stage.

The approach suggests that a system needs to be (1) described in a feature-ready form, (2) features should be described as revisions of *this* feature-ready form, and finally (3) interactions can be searched for at the various levels of mutability subdivision of the system with respect to the requirements that can be formulated in the feature-ready form. Search for interactions can be carried out in the form of verification of properties, which are either associated with the entire system, or with its various mutable parts, just like with feature constructs in general. The most decisive part of the job is, of course, to obtain the feature-ready description of the system. We believe that automating this cannot be done for the sole reasons of verifying, let alone for finding interactions. It should rather be part of the overall design process of the basic system. Being relevant to the partitioning of the work on the design and, possibly, of the implementation of a system, tool support for maintaining a feature-ready form should be part of a respective specification language. A conclusion that can be made here is that the capacity to describe feature integration is plausible not only for programming and specification languages meant for the specialist user, but also for intermediate languages that can possibly combine descriptions of diverse original forms that target diverse forms of implementation: very roughly speaking, both hardware and software. The gap between maintaining and obtaining a feature-ready form is a problem which the approach reveals as a very important one, if maximal automation is to be sought in all possible steps. The feature-readiness of a system can be decisive for its lifetime and creating feature-readiness is a development stage that developers are (ideally) well aware of and give it priority. Again, the flexibility in describing the existing practices of creating feature-readiness, rather than proposing particular architectures and then advocating the use of a corresponding system of techniques and tools, is the goal of this work. Further bringing the correspondences between such architectures to a level of precision can possibly contribute to understanding them and deciding how to enhance some of them or use them in parallel.

Acknowledgements

The authors are grateful to Alan Sexton, Dang Van Hung and Wang Yi for a number of discussions on the topic of this paper during its preparation.

References

[1] DANG VAN HUNG AND WANG JI. On The Design of Hybrid Control Systems Using Automata Models. *FST TCS 1996*, LCNS 1180, Springer, 1996, pp. 156-167.

[2] CALDER, M. AND E. MAGILL, *Feature Interactions in Telecommunications and Software Systems VI*, IOS Press, 2000.

[3] GUELEV. D. P. AND DANG VAN HUNG. Prefix and Projection onto State in Duration Calculus. *Electronic Notes in Theoretical Computer Science* 65, No 6, 2002, http : //www.elsevier.nl/locate/entcs/volume65.html, 19 p.

[4] GILMORE, S. AND M. D. RYAN (EDS.), *Language Constructs for Describing Features*, Springer-Verlag, 2001.

[5] GUELEV, D. P. A Complete Fragment of Higher-Order Duration μ-Calculus. *Proceedings of FST TCS 2000*. LNCS 1974, Springer Verlag, 2000. Available as Technical Report 195 from http : //www.iist.unu.edu/newrh/III/1/page.html .

[6] HANSEN, M. R. AND ZHOU CHAOCHEN. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9, 1997, pp. 283-330.

[7] MCMILLAN, K. *SMV documentation postscript versions.* http : //www-cad.eecs.berkeley.edu/~kenmcmil/psdoc.html . Accessed in February, 2002.

[8] PANDYA, P. K. Some extensions to Mean-Value Calculus: Expressiveness and Decidability. *Proceedings of CSL'95*. LNCS 1092, 1995, pp. 434-451.

[9] PLATH, M. C. AND M. D. RYAN. The Feature Construct for SMV: Semantics, In [2], pp. 129-144.

[10] PLATH, M. C. AND M. D. RYAN. Feature Integration Using a Feature Construct. *Science of Computer Programming*, 41(1), pp. 53-84, 2001. Available from http : //www.cs.bham.ac.uk/~mdr/research/papers/index.html .

[11] RAVN, A. P. *Design of Embedded Real-Time Computing Systems.* Dr. techn. dissertation. Technical report 1995-170, Technical University of Denmark, 1995.

[12] SAMBORSKI, D. Stack service model. *Language Constructs for Describing Features.* In [4], pp. 177-196.

[13] WANG HANPIN AND XU QIWEN *Temporal Logics over Infinite Intervals.* Technical Report 158, UNU/IIST, P.O.Box 3058, Macau, May 1999. To appear in *Discrete Applied Mathematics.*

[14] ZHOU CHAOCHEN AND M. R. HANSEN. An Adequate First Order Interval Logic. *COMPOS'97*, LNCS 1536, Springer, 1998, pp. 584-608.

[15] ZHOU CHAOCHEN, C. A. R. HOARE AND A. P. RAVN. A Calculus of Durations. *Information Processing Letters*, 40(5), pp. 269-276, 1991.