Spinal Test Suites for Software Product Lines

Harsh Beohar
Center for Research on Embedded Systems
Halmstad University, Sweden
harsh.beohar@hh.se

Mohammad Reza Mousavi
Center for Research on Embedded Systems
Halmstad University, Sweden
m.r.mousavi@hh.se

A major challenge in testing software product lines is efficiency. In particular, testing a product line should take less effort than testing each and every product individually. We address this issue in the context of input-output conformance testing, which is a formal theory of model-based testing. We extend the notion of conformance testing on input-output featured transition systems with the novel concept of spinal test suites. We show how this concept dispenses with retesting the common behavior among different, but similar, products of a software product line.

1 Introduction

1.1 Motivation

Testing and debugging are labor-intensive parts of software development. In particular, testing a software product line is extremely time- and resource-consuming due to the various configurations of products that are derivable from the product line. In order to manage the complexity, the test process of a software product line must be efficiently coordinated: common features ought to be tested once and for all and only specific variation points of various configurations should be tested separately.

Model-based testing is an approach to structure the test process by exploiting test models. Input-output conformance testing (ioco) [24] is a formalization of model-based testing employing input-output labeled transition systems as models. In the past, we extended the formal definition ioco to the setting of software product lines [3], by exploiting input-output featured transition systems. In this paper, we define a theoretical framework, which serves as the first step towards an efficient discipline of conformance model-based testing for software product lines.

To this end, we define the notion of *spinal test suite*, which allows one to test the common features once and for all, and subsequently, only focus on the specific features when moving from one product configuration to another. We show that spinal test suites are exhaustive, i.e., reject each and every non-conforming implementation under test, when the implementation satisfies the *orthogonality criterion*. This is a rather mild criterion, which implies that old features are not capable of disabling any enabled behavior from the new features on their own and without involving any interaction with the new feature's components.

1.2 Running example

To motivate various concepts throughout the paper, we use the following running example. Consider an informal description of a cruise controller, present in contemporary cars. The purpose of a cruise controller is to automatically maintain the speed of the car as specified by the driver. We denote the basic feature of a cruise controller by cc. Cruise controllers also have an optional feature, called collision avoidance controller (cac), whose task is to react to any obstacle detected ahead of the car within a danger

Submitted to:
MBT 2014

zone. In case the collision avoidance feature is included in a cruise controller and an obstacle is detected, the engine power is regulated using an emergency control algorithm.

1.3 Organization

The rest of this paper is structured as follows. In Section 2, we recall the formal definitions regarding models, product derivation and conformance testing. In Section 3, we define the notion of spinal test suite, which is a compact test suite for the "new" features with respect to an already tested product (or a set of features). In Section 4, we study the exhaustiveness of the spinal test suites: we show that spinal test suites are in general non-exhaustive, but this can be remedied by requiring mild conditions on the implementation under test. In Section 5, we sketch the context of this research. In Section 6, we conclude the paper and outline the direction of our ongoing research.

2 Background

2.1 Input-output featured transition systems

Feature diagrams [13, 22] have been used to model variability constraints in SPLs using a graphical notation. However, it is well known that feature diagrams only specify the structural aspects of variability and they should be complemented with other models in order to specify the behavioral aspects [7]. To this end, we describe the behavior of a software product line using an input-output featured transition system (IOFTS) [3], defined and explained below.

Let F be the set of features (extracted from a feature diagram) and $\mathbb{B} = \{\top, \bot\}$ be the set of Boolean constants; we denote by $\mathbb{B}(F)$ the set of all propositional formulae generated by interpreting the elements of the set F as propositional variables. For instance, in our running example, formula $\mathsf{cc} \wedge \neg \mathsf{cac}$ asserts the presence of cruise controller and the absence of collision avoidance controller. We let φ, φ' range over the set $\mathbb{B}(F)$.

Definition 1. *A* input-output featured transition system *(IOFTS) is a 6-tuple* $(S, s, A_\tau, F, T, \Lambda)$*, where*

1. *S is the set of* states*,*
2. *$s \in S$ is the* initial state*,*
3. *$A_\tau = A_I \uplus A_O \uplus \{\tau\}$ is the set of* actions*, where A_I and A_O are disjoint sets of* input *and* output *actions, respectively, and τ is the silent (internal) action,*
4. *F is a set of* features*,*
5. *$T \subseteq S \times A_\tau \times \mathbb{B}(F) \times S$ is the* transition relation *satisfying the following condition (for every $s_1, s_2 \in S, a \in A_\tau, \varphi, \varphi' \in \mathbb{B}(F)$):*

$$(s_1, a, \varphi, s_2) \in T \wedge (s_1, a, \varphi', s_2) \in T \Rightarrow \varphi = \varphi',[1]$$

6. *$\Lambda \subseteq \{\lambda : F \to \mathbb{B}\}$ is a set of* product configurations.

We write $s \xrightarrow{a}_{\varphi} s'$ to denote an element $(s, a, \varphi, s') \in T$ and drop the subscript φ whenever it is clear from the context. Graphically, we denote the initial state of an IOFTS by an incoming arrow with no

[1]Here, by $\varphi = \varphi'$ we assert that φ and φ' are syntactically equivalent.

source state and we refer to an IOFTS by its initial state. Following the standard notation, we denote the *reachability* relation by $\twoheadrightarrow \subseteq S \times A^* \times S$, which is inductively defined as follows:

$$\frac{}{s \overset{\varepsilon}{\twoheadrightarrow} s} \quad \frac{s \overset{\sigma}{\twoheadrightarrow} s', s' \xrightarrow{\tau} s''}{s \overset{\sigma}{\twoheadrightarrow} s''} \quad \frac{s \overset{\sigma}{\twoheadrightarrow} s', s' \xrightarrow{a} s'', a \neq \tau}{s \overset{\sigma a}{\twoheadrightarrow} s''}.$$

Furthermore, the set of *reachable* states from a state s is denoted by $\mathrm{Reach}(s) = \{s' \mid \exists_\sigma\ s \overset{\sigma}{\twoheadrightarrow} s'\}$.

Example 1. *Consider the IOFTS of a cruise controller, drawn in Figure 1, where inputs and outputs are prefixed with symbols* ? *and* !, *respectively. (Note that* ? *and* ! *are* not part *of the action names and are left out when the type of the action is irrelevant or clear from the context.) The regulate action, indicated*

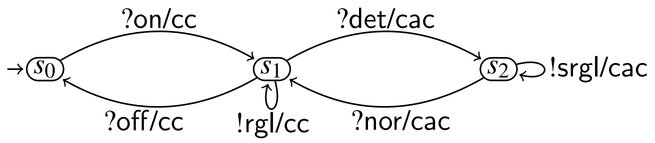

Figure 1: IOFTS of the cruise controller.

by rgl, *regulates the engine power of the car when the cruise controller is activated. Furthermore, when* cac *is included in a product, some additional behavior may emerge. Namely, while the cruise controller is on, if an object is detected within a danger zone, then the cruise controller regulates the engine power in a safe manner denoted by* srgl. *When the sensor signals a normal state, the cruise controller returns to the normal regulation regime. (For a realistic case study of a cruise controller and its formal model, we refer to [15].)*

2.2 Product derivation operators

In [3], we introduced a family of product derivation operators (parameterized by feature constraints), which project the behavior of an IOFTS into another IOFTS representing a selection of products (a product sub-line).

Definition 2. *Given a feature constraint φ and an IOFTS $(S, s, A_\tau, F, T, \Lambda)$, the projection operator Δ_φ induces an IOFTS $(S', \Delta_\varphi(s), A_{\tau\delta}, F, T', \Lambda')$, where*

1. *$S' = \{\Delta_\varphi(s') \mid s' \in S\}$ is the set of states,*
2. *$\Delta_\varphi(s)$ is the initial state,*
3. *$A_{\tau\delta} = A_\tau \uplus \{\delta\}$ is the set of actions, where δ is the special action label modeling quiescence [24],*
4. *T' is the smallest relation satisfying:*

$$\frac{s \xrightarrow{a}_{\varphi'} s' \quad \exists_\lambda\ (\lambda \in \Lambda \wedge \lambda \models (\varphi \wedge \varphi'))}{\Delta_\varphi(s) \xrightarrow{a}_{\varphi \wedge \varphi'} \Delta_\varphi(s')} \qquad (1)$$

$$\frac{\bar{\Lambda} = \{\lambda \in \Lambda \mid \lambda \models \varphi \wedge \mathbf{Q}(s, \lambda)\} \quad \bar{\Lambda} \neq \emptyset}{\Delta_\varphi(s) \xrightarrow{\delta}_{\varphi \wedge (\bigvee_{\lambda \in \bar{\Lambda}} \lambda)} \Delta_\varphi(s)} \qquad (2)$$

where the predicate $\mathbf{Q}(s, \lambda)$ is defined as

$$\forall_{s', a, \varphi'}\ \left(s \xrightarrow{a}_{\varphi'} s' \wedge a \in A_O \cup \{\tau\}\right) \Rightarrow \lambda \not\models \varphi'.$$

5. $\Lambda' = \{\lambda \in \Lambda \mid \lambda \models \varphi\}$ *is the set of product configurations.*

In the above-given rules $\lambda \models \varphi$, denotes that valuation λ of features satisfies feature constraint φ. Intuitively, rule (1) describes the behavior of those valid products that satisfy the feature constraint φ in addition to the original annotation of the transition emanating from s. Rule (2) models quiescence (the absence of outputs and internal actions) from the state $\Delta_\varphi(s)$. Namely, it specifies that the projection with respect to φ is quiescent, when there exists a valid product λ that satisfies φ and is quiescent, i.e., cannot perform any output or internal transition. Quiescence at state s for a feature constraint λ is formalized using the predicate $\mathbf{Q}(s,\lambda)$, which states that from state s there is no output or silent transition with a constraint satisfied by λ. In the conclusion of the rule, a δ self-loop is specified and its constraint holds when φ holds and at least the feature constraint of one quiescent valid product holds. This ability to observe the absence of outputs (through a timeout mechanism) is crucial in defining the input-output conformance relation between a specification and an implementation [3].

Example 2. *Consider the feature constraint* $\varphi = \mathsf{cc} \wedge \neg\mathsf{cac}$. *The IOFTS generated by projecting the IOFTS of cruise controller (in Figure 1) using feature constraint* φ *is depicted in Figure 2. As mentioned before, this represents the product that has the basic cruise controller functionality but does not contain collision avoidance controller.*

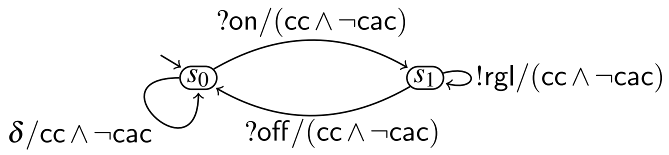

Figure 2: Cruise controller after projecting with feature constraint $\mathsf{cc} \wedge \neg\mathsf{cac}$.

In the sequel, we use the phrase "a feature specification $\Delta_\varphi(s)$" to refer to the following IOFTS:

$$(\mathrm{Reach}(\Delta_\varphi(s)), \Delta_\varphi(s), A_{\tau\delta}, F, T, \Lambda).$$

We interpret the original IOFTS of Definition 1 as $\Delta_\top(s_0)$; this has the implicit advantage of always including quiescence in appropriate states.

2.3 Input-output conformance

The input-output conformance (ioco) testing theory [24] formalizes model-based testing in terms of a conformance relation between the states of a model (expressed as an input-output transition system) and an implementation under test (IUT). Note that the ioco theory is based on the *testing assumption* that the behavior of the IUT can be expressed by an input-output transition system, which is unknown to the tester.

The conformance relation can be checked by constantly providing the SUT with inputs that are deemed relevant by the model and observing outputs from the SUT and comparing them with the possible outputs prescribed by the model. In the following, we recall such an *extensional* definition of ioco, extended to software product lines in [3]. An equivalent *intensional* definition of ioco that relies on comparing the traces of the underlying IOFTS was also given in [3], but for the purpose of this paper we only work with the extensional definition. (After all, the extensional definition is the one that is supposed to be applied in practice.)

We begin with a notion of *suspension traces* generated by an IOFTS. Informally, a suspension trace is a trace that may contain the action δ denoting quiescence [24].

Definition 3. *The set of* suspension traces *of a feature specification* $\Delta_\varphi(s)$*, denoted by* $\mathit{Straces}(\Delta_\varphi(s))$ *is defined as:* $\{\sigma \in A_\delta{}^* \mid \exists_{s'}\ \Delta_\varphi(s) \overset{\sigma}{\twoheadrightarrow} \Delta_\varphi(s')\}$.

For example, in the IOFTS of Example 2, δ?on!rgl is a suspension trace emanating from the initial state s_0. Next, we define the notion of test suite, which summarizes all possible test cases that can be generated from a feature specification.

Definition 4. *The* test suite *for an IOFTS* $(\mathit{Reach}(\Delta_\varphi(s)), \Delta_\varphi(s), A_{\tau\delta}, F, T, \Lambda)$*, dennoted by* $\mathscr{T}(s,\varphi)$*, is the IOFTS* $(\mathbf{X} \cup \{\mathbf{pass},\mathbf{fail}\}, \mathbf{X}_0, A_\delta, F, T', \Lambda)$*,, where*

1. $\mathbf{X} = \big\{\big(\{s' \mid \Delta_\varphi(s) \overset{\sigma}{\twoheadrightarrow} \Delta_\varphi(s')\}, \sigma\big) \mid \sigma \in \mathit{Straces}(s)\big\}$ *is the set of intermediate states and* $\{\mathbf{pass},\mathbf{fail}\}$ *is the set of* verdict states *[24],*
2. $\mathbf{X}_0 = \{(\{s' \mid \Delta_\varphi(s) \overset{\varepsilon}{\twoheadrightarrow} \Delta_\varphi(s')\}, \varepsilon)\}$ *is the initial state of the test suite,*
3. $A_\delta = A \uplus \{\delta\}$ *is the set of actions, and*
4. *the transition relation* T' *is defined as the smallest relation satisfying the following rules.*

$$\frac{(X,\sigma),(Y,\sigma a) \in \mathbf{X}}{(X,\sigma) \xrightarrow{a}_\varphi (Y,\sigma a)} \quad (3) \qquad \frac{\begin{array}{c} a \in A_O \cup \{\delta\} \\ (X,\sigma) \xrightarrow{a}_\varphi (Y,\sigma') \end{array}}{(X,\sigma) \xrightarrow{a}_\varphi \mathbf{pass}} \quad (4)$$

$$\frac{\begin{array}{c} a \in A_O \cup \{\delta\} \\ (X,\sigma) \not\xrightarrow{a}_\varphi \mathbf{pass} \end{array}}{(X,\sigma) \xrightarrow{a}_\varphi \mathbf{fail}} \quad (5) \qquad \frac{a \in A_O \cup \{\delta\}}{\begin{array}{c} \mathbf{pass} \xrightarrow{a}_\varphi \mathbf{pass} \\ \mathbf{fail} \xrightarrow{a}_\varphi \mathbf{fail} \end{array}} \quad (6)$$

Intuitively, the test suite for a feature specification is an IOFTS (possibly with an infinite number of states), which contains all the possible test cases that can be generated from the feature specification. Rule (3) states that if X and Y are nonempty sets of reachable states from s (under feature restriction φ) with the suspension traces σ and σa, respectively, then there exists a transition of the form $(X,\sigma) \xrightarrow{a}_\varphi (Y,\sigma a)$ in the test suite. Rules (4) and (5) model, respectively, the successful and the unsuccessful observation of outputs and quiescence. Note that input actions are not included in rules (4) and (5) because the implementation is assumed to be input-enabled [24]; hence, they are already covered by rule (3). Rule (6) states that the verdict states contain a self-loop for each and every output action, as well as for quiescence.

Example 3. *The test suite for the IOFTS of Example 2 is (partially) depicted in Figure 3.*

A reader familiar with the original ioco theory [24] will immediately notice that our definition of a test suite (Definition 4) is nonstandard. In particular, a test suite is defined as a set of test cases (i.e., input-output transition systems with certain restrictions) with finite number of states in [24]; whereas we represent a test suite by an IOFTS, possibly with an infinite number of states. To this end, we define a test case to be a finite projection of a test-suite with the additional restriction that at each moment of time at most one input can be fed into the system under test (see [3] for a formal definition). As a result, our test cases are structurally similar to Tretmans' formulation of the test cases, by which we mean that:

- a test case is always deterministic,

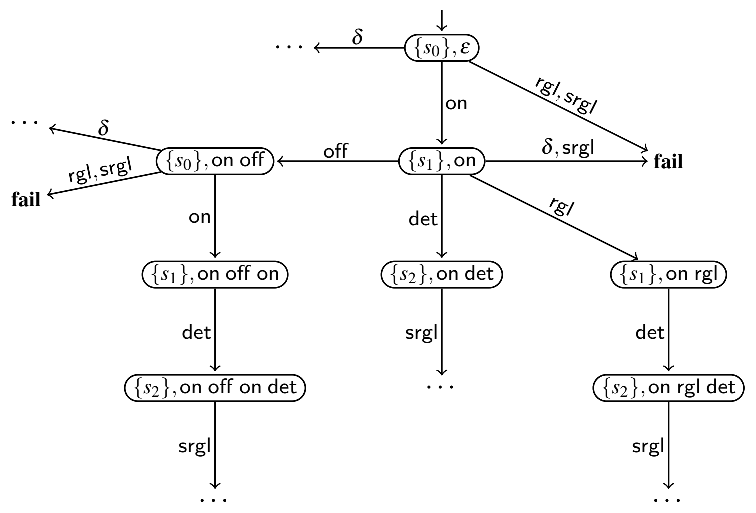

Figure 3: The test suite of the cruise controller example.

- a test case is always input enabled, and
- a test case has no cycles except those in the verdict states **pass** and **fail**.

Another notable difference, that is key to define the concepts of Section 3, is that states of a test suite (or test case) carry some mathematical structure, whereas the states of a test case in [24] are abstract and carry no structure.

Next, we define a *synchronous observation* operator $\rceil\!|$ that allows us to model a test run on an implementation (cf. [24]). This is defined over a test suite and an IOFTS (the intended implementation) as follows. (Note that the calligraphic letters $\mathscr{X}, \mathscr{Y}$ in the following rules range over the states of a test suite.)

$$\frac{\mathscr{X} \xrightarrow{a} \mathscr{Y} \quad \Delta_\varphi(s) \xrightarrow{a} \Delta_\varphi(s') \quad a \in A_\delta}{\mathscr{X}\rceil\!|\Delta_\varphi(s) \xrightarrow{a}_\top \mathscr{Y}\rceil\!|\Delta_\varphi(s')} \quad (7) \qquad \frac{\Delta_\varphi(s) \xrightarrow{\tau} \Delta_\varphi(s')}{\mathscr{X}\rceil\!|\Delta_\varphi(s) \xrightarrow{\tau}_\top \mathscr{X}\rceil\!|\Delta_\varphi(s')} \quad (8)$$

Having defined the notion of synchronous observation, we can now define what it means for a feature specification to pass s test suite. Informally, a feature specification passes a test suite if and only if no trace of the synchronous observation of the test suite and the feature specification leads to the **fail** verdict state.

Definition 5. *Let* $\mathbf{X}_0$ *be the initial state of a test suite* $\mathscr{T}(s, \varphi)$*. A feature specification* $\Delta_{\varphi'}(s')$ passes *the test suite* $\mathscr{T}(s, \varphi)$ *iff*

$$\forall_{\sigma \in A_\delta{}^*, \mathscr{X}, s''} \ \mathbf{X}_0\rceil\!|\Delta_{\varphi'}(s') \overset{\sigma}{\twoheadrightarrow} \mathscr{X}\rceil\!|\Delta_{\varphi'}(s'') \ \Rightarrow \ \mathscr{X} \neq \mathbf{fail}.$$

Thexxxxxxxxxxx implementation $\Delta_{\varphi'}(s')$ conforms *to the specification* $\Delta_\varphi(s)$ *iff* $\Delta_{\varphi'}(s')$ *passes the test suite* $\mathscr{T}(s, \varphi)$.

3 Spinal test suite

As mentioned in the introduction, one of the challenges in testing a software product line is to minimize the test effort. The idea pursued in this section is to organize the test process of a product line incrementally. This is achieved by reusing the test results of an already tested product to test a product with similar features, thereby dispensing with the test cases targeted at the common features. To this end, we introduce the notion of *spinal test suite*, which prunes away the behavior of a specified set of features from an *abstract* test suite $\mathscr{T}(s,\varphi)$ with respect to a *concrete* test suite $\mathscr{T}(s,\lambda)$ of the already tested product λ; the spinal test suite is only defined when λ is valid w.r.t. φ, i.e., $\lambda \models \varphi$. The latter constraint means that the concrete product builds upon the already-tested features in the abstract test suite.

Notably, which behavior has to be pruned from an abstract test suite is crucial in defining a spinal test suite. One way to address this situation is *by allowing only those reachable states in the abstract test suite from which a new behavior relative to the already tested product emanates*. However, without any formal justification, we claim that such a strategy will not reduce the effort to test new behavior with respect to the already tested product.

For example, consider the test suite depicted in Figure 3 and suppose we have already tested the cruise controller without collision avoidance feature and now are interested in the correct implementation of the collision avoidance feature. By following the aforementioned strategy of pruning, none of the following states $(\{s_1\},\mathsf{on}),(\{s_1\},\mathsf{on\ off\ on}),\cdots$ will be removed because the event det is enabled from each of these states. On the other hand, since we know that cruise controller without collision avoidance feature was already tested, it is safe to consider the new suspension traces (or testing experiments) from only one state in $\{(\{s_1\},\mathsf{on}),(\{s_1\},\mathsf{on\ off\ on}),\cdots\}$.

Definition 6. *Let $\mathbf{X}_0$ be the initial state of a test suite $\mathscr{T}(s,\varphi)$. A path $\mathbf{X}_0 \overset{\sigma}{\twoheadrightarrow} (X,\sigma)$ is a* spine *of a path $\mathbf{X}_0 \overset{\sigma'}{\twoheadrightarrow} (X,\sigma')$, denoted by $\sigma\dagger\sigma'$, when σ is a sub-trace of σ' (obtained by removing zero or more action from σ') and no two states visited during the trace σ have the same X-component; this is formalized by the predicate $\mathbf{bt}(X,\sigma)$, defined below:*

$$\forall_{\sigma_1,\sigma_2,\sigma_3,Y,Z}\ \big(\mathbf{X}_0 \overset{\sigma_1}{\twoheadrightarrow} (Y,\sigma_1) \overset{\sigma_2}{\twoheadrightarrow} (Z,\sigma_2) \overset{\sigma_3}{\twoheadrightarrow} (X,\sigma) \wedge \sigma_2 \neq \varepsilon \wedge \sigma = \sigma_1\sigma_2\sigma_3\big) \Rightarrow Y \neq Z.$$

Furthermore, we let $\mathbf{bt}(\mathbf{X}_0) = \top$.

Example 4. *Recall the feature specification given $\Delta_\varphi(s_0)$ in Example 2, where $\varphi = \mathsf{cc} \wedge \neg\mathsf{cac}$. Since collision avoidance controller is an optional feature, we know that there exists a product configuration λ with $\lambda(\mathsf{cc}) = \top$ and $\lambda(\mathsf{cac}) = \bot$. Then, the path labelled "*on*" (in the test suite drawn in Figure 3) is a spine of the path labelled "*on off on*" because they both reach to a common X-component $\{s_1\}$ in the test suite and $\mathbf{bt}(\{s_1\},\mathsf{on}) = \top$.*

Definition 7. *Let $(\mathbf{X} \cup \{\mathbf{pass},\mathbf{fail}\},\mathbf{X}_0,A_\delta,F,T,\Lambda)$ be a test suite $\mathscr{T}(s,\varphi)$ and let λ be a product such that $\lambda \models \varphi$. Then a* spinal test suite *with respect to a product λ, denoted by $\mathscr{S}(\varphi,\lambda)$, is an IOFTS $(\mathbf{X} \cup \{\mathbf{pass},\mathbf{fail}\},\mathbf{X}_0,A_\delta,F,T',\Lambda')$, where*

1. *The set of non-verdict states $\mathbf{X}$ is defined as $\mathbf{X}' \cup \mathbf{X}''$, where*

$$\mathbf{X}' = \{(X,\sigma) \in \mathbf{X}_s^\varphi \mid \sigma \in Straces(\Delta_\lambda(s)) \wedge \mathbf{bt}(X,\sigma)\}$$
$$\mathbf{X}'' = \{(Y,\sigma\sigma') \in \mathbf{X}_s^\varphi \mid \sigma\sigma' \notin Straces(\Delta_\lambda(s)) \wedge \exists_{(X,\sigma)\in\mathbf{X}'}\ (X,\sigma) \overset{\sigma'}{\twoheadrightarrow} (Y,\sigma\sigma')\}.$$

2. *The set of transition relations T' is defined as*

$$T' = \{(\mathscr{X},a,\mathscr{Y}) \in T \mid \mathscr{X},\mathscr{Y} \in \mathbf{X}\}.$$

3. *The set of product configurations $\Lambda' = \Lambda \setminus \{\lambda\}$.*

Intuitively, Condition 1 defines $\mathbf{X}'$ to be a set of non-verdict states of the form (X, σ) such that σ is a suspension trace of the already tested product $\Delta_\lambda(s)$ and the predicate $\mathbf{bt}(X, \sigma)$ holds; whereas, $\mathbf{X}''$ is the set of non-verdict states reachable from a state in $\mathbf{X}'$ by a trace that is not a suspension trace of the tested product $\Delta_\lambda(s)$. Condition 2 and 3 are self-explanatory.

As an example, the spinal test suite generated from the test suite in Figure 3 is partially drawn in Figure 4.

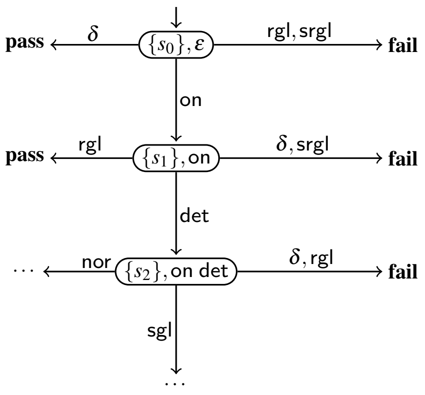

Figure 4: Spinal test suite of the cruise controller

4 Exhaustiveness of Spinal Test Suites

The spinal test suite $\mathscr{S}(\varphi, \lambda)$ contains the spines of those paths from the test suite $\mathscr{T}(s, \varphi)$ that lead to new behavior w.r.t. to the already-tested product λ. Next, we show that the spinal test suite $\mathscr{S}(\varphi, \lambda)$ is not necessarily exhaustive for an arbitrary implementation under test, i.e., it may have *strictly* less testing power than the test suite $\mathscr{T}(s, \varphi)$. We show this through the following example.

Example 5. *Consider an implementation of a cruise controller with control avoidance feature modelled as an IOFTS, depicted in Figure 5. Clearly, this implementation is a faulty one as the action* 'rgl' *must*

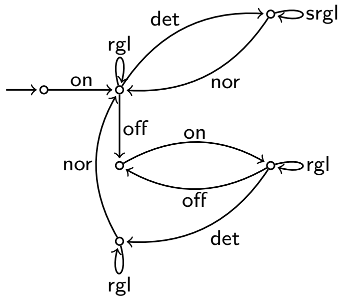

Figure 5: A faulty implementation of the cruise controller with control avoidance.

be prohibited after detecting an obstacle, i.e., after executing the transition labelled 'det'*.*

As soon as we place the test suite (Figure 3) in parallel ($\rceil|$) with the above implementation, we observe that the following synchronous interactions on.off.on.det.rgl *between the test suite and the above* *implementation leads to the* **fail** *verdict state. However, note that the aforementioned fault in the implementation cannot be detected while interacting with the spinal test suite (Figure 4) because there are no transitions labelled with* off *in the spinal test suite. Thus, a spinal test suite $\mathscr{S}(\varphi,\lambda)$ has strictly less testing power than the test suite $\mathscr{T}(s,\varphi)$.*

Next, we explore when a spinal test suite $\mathscr{S}(\varphi,\lambda)$ (where $\lambda \models \varphi$) together with a concrete test suite $\mathscr{T}(s,\lambda)$ have the same testing power as the abstract test suite $\mathscr{T}(s,\varphi)$.

Definition 8. *Let $\lambda \models \varphi$. A feature specification $\Delta_{\varphi'}(s')$ is* orthogonal *w.r.t $\Delta_\varphi(s)$ and the product λ iff for every $s_1, \sigma' \in Straces(\Delta_\lambda(s))$, $\sigma'' \neq \varepsilon$, and $\sigma'\sigma'' \notin Straces(\Delta_\lambda(s))$ we have*

$$\Delta_{\varphi'}(s') \xrightarrow{\sigma'\sigma''}\!\!\!\!\!\rightarrow \Delta_{\varphi'}(s_1) \Rightarrow \exists_{s_2,\sigma}\ \Delta_{\varphi'}(s') \xrightarrow{\sigma\sigma''}\!\!\!\!\!\rightarrow \Delta_{\varphi'}(s_2) \wedge \sigma \dagger \sigma'.$$

Example 6. *Recall the feature specification $\Delta_\varphi(s_0)$ and the product λ (which omits the control avoidance feature) from Example 4. Note that the implementation given in Figure 5 is not orthogonal w.r.t the feature specification $\Delta_\varphi(s_0)$ and the product λ because the underlined subsequence in "*on off on $\underline{\text{det rgl}}$*" cannot be extended with the spine sequence* on.

In the remainder of this section, we prove the main result (Theorem 1) of this section that an orthogonal implementation passes the test suite $\mathscr{T}(s,\varphi)$ whenever it passes the concrete test suite $\mathscr{T}(s,\lambda)$ and the spinal test suite $\mathscr{S}(\varphi,\lambda)$.

Lemma 1. *Let $\mathbf{X}_0$ be the initial state of a test suite $\mathscr{T}(s,\varphi)$ and let λ be a product with $\lambda \models \varphi$. If $\mathbf{X}_0 \xrightarrow{\sigma'\sigma''}\!\!\!\!\!\rightarrow$ **fail**, $\sigma' \in Straces(\Delta_\lambda(s))$, $\sigma'\sigma'' \notin Straces(\Delta_\lambda(s))$, and $\sigma \dagger \sigma'$ then $\mathbf{X}_0 \xrightarrow{\sigma\sigma''}\!\!\!\!\!\rightarrow$ **fail**.*

Proof sketch. Let us first decompose the sequence of transitions $\mathbf{X}_0 \xrightarrow{\sigma'\sigma''}\!\!\!\!\!\rightarrow$ **fail** as $\mathbf{X}_0 \xrightarrow{\sigma'}\!\!\!\!\!\rightarrow (X,\sigma') \xrightarrow{\sigma''}\!\!\!\!\!\rightarrow$ **fail**, for some X. Then by definition of a spine path we get $\mathbf{X}_s^\varphi \xrightarrow{\sigma}\!\!\!\!\!\rightarrow (X,\sigma)$. Next, it is straightforward to show by induction on σ'' that $(X,\sigma) \xrightarrow{\sigma''}\!\!\!\!\!\rightarrow$ **fail**, whenever $(X,\sigma') \xrightarrow{\sigma''}\!\!\!\!\!\rightarrow$ **fail**. □

Theorem 1. *Let $\Delta_{\varphi'}(s')$ be orthogonal w.r.t. to $\Delta_\varphi(s)$ and λ. If $\Delta_{\varphi'}(s')$ pass the test suites $\mathscr{T}(s,\lambda)$ and $\mathscr{S}(\varphi,\lambda)$, then $\Delta_{\varphi'}(s')$ passes the test suite $\mathscr{T}(s,\varphi)$.*

Proof. Let $\mathbf{X}_0$ be the initial state of the test suite $\mathscr{T}(s,\varphi)$. We will prove this theorem by contradiction. Let $\Delta_{\varphi'}(s')$ pass the test suites $\mathscr{T}(s,\lambda)$ and $\mathscr{S}(\varphi,\lambda)$. Suppose $\Delta_{\varphi'}(s')$ fails in passing the test suite $\mathscr{T}(s,\varphi)$. Then, there exists the following sequences of transitions $\mathbf{X}_0 \xrightarrow{\sigma}\!\!\!\!\!\rightarrow$ **fail** and $\Delta_{\varphi'}(s') \xrightarrow{\sigma}\!\!\!\!\!\rightarrow \Delta_{\varphi'}(s_1')$ (for some σ, s_1') in the test suite $\mathscr{T}(s,\varphi)$ and the feature specification $\Delta_{\varphi'}(s')$. Now there are two possibilities:

1. Either, $\sigma \in \text{Straces}(\Delta_\lambda(s))$. Then, the feature specification $\Delta_{\varphi'}(s')$ fails to pass the test suite $\mathscr{T}(s,\lambda)$. Hence, a contradiction.

2. Or, $\sigma \notin \text{Straces}(\Delta_\lambda(s))$. Then, the sequence of transitions $\mathbf{X}_0 \xrightarrow{\sigma}\!\!\!\!\!\rightarrow$ **fail** can be decomposed in the following way: $\mathbf{X}_0 \xrightarrow{\sigma_1\sigma_2}\!\!\!\!\!\rightarrow$ **fail** with $\sigma = \sigma_1\sigma_2$ and $\sigma_1 \in \text{Straces}(\Delta_\lambda(s))$.

   Since the feature specification $\Delta_{\varphi'}(s')$ is orthogonal w.r.t. $\Delta_\varphi(s)$ and λ, we have $\exists_{s_2',\sigma_1'}\ \Delta_{\varphi'}(s') \xrightarrow{\sigma_1'\sigma_2}\!\!\!\!\!\rightarrow \Delta_{\varphi'}(s_2') \wedge \sigma_1' \dagger \sigma_1$. Then, by applying Lemma 1 we get the following path in the spinal test suite: $\mathbf{X}_0 \xrightarrow{\sigma_1'\sigma_2}\!\!\!\!\!\rightarrow$ **fail**. Thus, $\Delta_{\varphi'}(s')$ fails to pass the spinal test suite $\mathscr{S}(\varphi,\lambda)$; hence, a contradiction. □

5 Related work

Various attempts have been made in formal and informal modeling of SPLs, on which [20, 6, 21, 9, 23] provide comprehensive surveys. By and large the literature can be classified into two categories: structural modeling and behavioral modeling techniques.

Structural models specify variability in terms of presence and absence of features (assets, artifacts) in various products and their mutual inter-relations. Behavioral models, however, concern the working of features and their possible interactions, mostly based on some form of finite state machines or labeled transition systems. The main focus in behavioral modeling of SPLs (cf. [2, 1, 7, 8, 11, 12, 16]) has been on formal specification of SPLs and adaptation of formal verification (mostly model checking) techniques to this new setting.

In addition, several testing techniques have been adapted to SPLs, of which [19, 14, 18, 10] provide recent overviews. Hitherto, most fundamental approaches to formal conformance testing [4] have not been adapted sufficiently to the SPL setting. The only exception that we are aware of is [17], which presents an LTS-based incremental derivation of test suites by applying principles of regression testing and delta-oriented modeling [5].

Although our work is based on input-output conformance testing, we envisage that the ideas pursued in this paper can be adapted to other fundamental theories of conformance testing, e.g., those based on finite state machines [25, 4].

6 Conclusions

In this paper, we introduced the notion of spinal test suites, which can be used in order to incrementally test different products of a software product line. A spinal test suite only tests the behavior induced by the "new" features and dispenses with re-testing the already-tested behavior, unless this is necessary in order to reach the behavior of the new features.

As future work, we intend to exploit this notion and establish a methodology of testing software product lines, by automatically detecting the optimal order of testing products, which leads to a minimal size of residual test suites (with respect to a given notion of model coverage). In order to effectively use the notion of spinal test suites, we would like to define syntactic criteria that guarantee orthogonality of features.

Acknowledgments

We thank the anonymous reviewers of MBT 2014 for their useful feedback.

References

[1] P. Asirelli, M. H. ter Beek, A. Fantechi & S. Gnesi (2011): *A Model-Checking Tool for Families of Services*. In R. Bruni & J. Dingel, editors: *Formal Techniques for Distributed Systems*, *Lecture Notes in Computer Science* 6722, Springer Berlin Heidelberg, pp. 44–58, doi:10.1007/978-3-642-21461-5_3.

[2] P. Asirelli, M. H. ter Beek, S. Gnesi & A. Fantechi (2011): *Formal Description of Variability in Product Families*. In E. Almeida, T. Kishi, C. Schwanninger, I. John & K. Schmid, editors: *Proc. of 15th International Software Product Line Conference*, IEEE, pp. 130–139, doi:10.1109/SPLC.2011.34.

[3] H. Beohar & M. R. Mousavi (2014): *Input-Output Conformance Testing Based on Featured Transition Systems*. In: *Proceedings of the the 29th Symposium On Applied Computing*, ACM Press. To appear, available from: `http://ceres.hh.se/mediawiki/images/b/b0/ Mousavi_svt_2014.pdf`.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker & A. Pretschner (2005): *Model-Based Testing of Reactive Systems*. *Lecture Notes in Computer Science* 3472, Springer Berlin Heidelberg, doi:10.1007/b137241.

[5] D. Clarke, M. Helvensteijn & I. Schaefer (2010): *Abstract delta modeling*. In E. Visser & J. Järvi, editors: *Proceedings of the 9th international conference on Generative programming and component engineering*, GPCE '10, ACM, NY, USA, pp. 13–22, doi:10.1145/1868294.1868298.

[6] A. Classen (2010): *Modelling with FTS: a Collection of Illustrative Examples*. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur. Available at `http://www.fundp.ac.be/pdf/publications/69416.pdf`.

[7] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay & J.-F. Raskin (2013): *Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking*. *IEEE Transactions on Software Engineering* 39(8), pp. 1069–1089, doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay & J.-F. Raskin (2010): *Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines*. In J. Kramer, J. Bishop, P. T. Devanbu & S. Uchitel, editors: *32nd International Conference on Software Engineering, ICSE '10* 1, ACM, New York, NY, USA, pp. 335–344, doi:10.1145/1806799.1806850.

[9] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid & A. Wasowski (2012): *Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches*. In U. W. Eisenecker, S. Apel & S. Gnesi, editors: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, ACM, New York, NY, USA, pp. 173–182, doi:10.1145/2110147.2110167.

[10] E. Engström & P. Runeson (2011): *Software Product Line Testing - A Systematic Mapping Study*. *Information & Software Technology* 53(1), pp. 2–13, doi:10.1016/j.infsof.2010.05.011.

[11] D. Fischbein, S. Uchitel & V. Braberman (2006): *A Foundation for Behavioural Conformance in Software Product Line Architectures*. In R. M. Hierons & H. Muccini, editors: *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, ACM, New York, NY, USA, pp. 39–48, doi:10.1145/1147249.1147254.

[12] A. Gruler, M. Leucker & K. Scheidemann (2008): *Modeling and Model Checking Software Product Lines*. In G. Barthe & F. S. de Boer, editors: *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, *Lecture Notes in Computer Science* 5051, Springer-Verlag, Berlin, Heidelberg, pp. 113–131, doi:10.1007/978-3-540-68863-1_8.

[13] K. Kang, S. Cohen, J. Hess, W. Novak & S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania. Available at `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231`.

[14] B. P. Lamancha, M. Polo & M. Piattini (2013): *Systematic Review on Software Product Line Testing*. In J. Cordeiro, M. Virvou & B. Shishkov, editors: *Software and Data Technologies*, *Comm. in Computer and Information Science* 170, Springer Berlin Heidelberg, pp. 58–71, doi:10.1007/978-3-642-29578-2_4.

[15] M.A. de Langen (2013): *Vehicle Function Correctness*. Masters Thesis, Eindhoven University of Technology. Available at `http://alexandria.tue.nl/extra1/afstversl/wsk-i/langen2013.pdf`.

[16] K. G. Larsen, U. Nyman & A. Wąsowski (2007): *Modal I/O Automata for Interface and Product Line Theories*. In: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 4421, Springer Berlin Heidelberg, pp. 64–79, doi:10.1007/978-3-540-71316-6_6.

[17] M. Lochau, I. Schaefer, J. Kamischke & S. Lity (2012): *Incremental Model-Based Testing of Delta-Oriented Software Product Lines*. In A. D. Brucker & J. Julliand, editors: *Tests and Proofs*, *Lecture Notes in Computer Science* 7305, Springer Berlin Heidelberg, pp. 67–82, doi:10.1007/978-3-642-30473-6_7.

[18] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida & Silvio Romero de Lemos Meira (2011): *A systematic mapping study of software product lines testing*. *Information and Software Technology* 53(5), pp. 407–423, doi:10.1016/j.infsof.2010.12.003.

[19] S. Oster, A. Wübbeke, G. Engels & A. Schürr (2011): *A Survey of Model-Based Software Product Lines Testing*. In J. Zander, I. Schieferdecker & P. J. Mosterman, editors: *Model-based Testing for Embedded Systems*, CRC Press, pp. 339–381.

[20] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo & K. Villela (2012): *Software diversity: state of the art and perspectives*. *International Journal on Software Tools for Technology Transfer* 14(5), pp. 477–495, doi:10.1007/s10009-012-0253-y.

[21] K. Schmid, R. Rabiser & P. Grünbacher (2011): *A Comparison of Decision Modeling Approaches in Product Lines*. In P. Heymans, K. Czarnecki & U. W. Eisenecker, editors: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, ACM, New York, NY, USA, pp. 119–126, doi:10.1145/1944892.1944907.

[22] P.-Y. Schobbens, P. Heymans & J.-C. Trigaux (2006): *Feature Diagrams: A Survey and a Formal Semantics*. In: *Proc. of the 14th IEEE International Conference on Requirements Engineering*, RE '06, IEEE Computer Society, Washington, DC, USA, pp. 136–145, doi:10.1109/RE.2006.23.

[23] M. Sinnema & S. Deelstra (2007): *Classifying Variability Modeling Techniques*. *Information & Software Technology* 49(7), pp. 717–739, doi:10.1016/j.infsof.2006.08.001.

[24] J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In R. M. Hierons, J. P. Bowen & M. Harman, editors: *Formal Methods and Testing*, chapter I, *Lecture Notes in Computer Science* 4949, Springer Berlin Heidelberg, pp. 1–38, doi:10.1007/978-3-540-78917-8_1.

[25] M. Yannakakis & D. Lee (1999): *Testing of Finite State Systems*. In G. Gottlob, E. Grandjean & K. Seyr, editors: *Computer Science Logic*, *Lecture Notes in Computer Science* 1584, Springer Berlin Heidelberg, pp. 29–44, doi:10.1007/10703163_3.