Specification-based Testing for Software Product Lines

Temesghen Kahsai* Markus Roggenbach* Bernd-Holger Schlingloff†

Abstract

In this paper, we develop a testing theory for specification-based software product line development. Starting with a framework for the evaluation of test cases with respect to formal specifications, we develop a notion of enhancement, which allows to re-use test cases in a horizontal systems development process. In such a process, more and more features are added to an existing software product. For specification-based testing, this means that the corresponding specifications are enhanced more and more, and that new test cases must be added to an existing test suite in order to test the additional features. We formally define an enhancement relation between CSP-CASL *specifications, describe a test evaluation method for software product lines based on* CSP-CASL *specifications, and prove several preservation results which allow to re-use test cases. We illustrate our approach with the example of a product line of remote control units for consumer products.*

1. Introduction

Today, very few software systems are developed from scratch; most systems are derived by extending or enhancing previous versions. Thus, traditional engineering approaches, in which a complete system is derived from a given set of informal or formal specifications, are only partially adequate. This holds in particular for *software product lines*, where a set of similar products is targeted. The CMU SEI defines a software product line to be a "set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [3]. Thus, the individual products in a product line have a similar "look-and-feel", however, they differ in that one product may offer more functionality than the other one. In a product line, there are low-end products with a basic set of features, specialized products for particular markets, and high-end products which combine many features. For the corresponding software development process, this means that the specification of an advanced product is developed by enhancement and combination of basic specifications.

For testing different elements of a product line, test cases for basic products should be reusable in advanced products. The design of test suites is a time-consuming engineering task; depending on the application domain, $30\% - 80\%$ of the overall software development costs are spent for testing. Main activities in testing are the construction of test cases, the execution of tests on a system-under-test (SUT), and the evaluation of test execution results. Often, the design of a test suite is *feature-oriented*: the test suite must be able to determine if exactly the specified features are implemented. That is, if some feature is required in the specification, the SUT must be able to exhibit the corresponding behaviour; vice versa, if the SUT shows some observable behaviour, then it must be part of some specified requirement. Thus, test suites are structured along the features of the specification. For each feature, there should be test cases present in the test suite to validate whether the feature has been correctly implemented.

In a software product line, the advanced product incorporates features from more basic versions. Even if all features of the basic products have been thoroughly tested, it is necessary to validate that these features still work correctly in the enhanced version. Usually the design and testing of the basic version is completed before the advanced version is begun; in this case, for all basic features elaborate test cases are available. It would be advantageous to be able to re-use these test cases in a test suite for the advanced product.

In this paper, we study the evaluation and preservation of test cases in specification-based software product line development. That is, we assume that test cases are developed with or without reference to a formal specification; test cases are assessed with respect to a particular specification, and evaluated with respect to a particular implementation (see Figure 1 below). Each software feature can be formalized in a separate specification; therefore, for each feature a test suite can be developed separately. Test suites can check for the presence or absence of a particular feature. The specification of a specific product of the product line consists of a combination of features. We argue that

*Swansea University

†Humboldt University Berlin Fraunhofer FIRST

This work was supported by EPSRC under the grant EP/D037212/1.

if this combination follows certain enhancement principles, then the assessment result of a test case is preserved from the separate feature to the combination. (On the other hand, if the enhancement principles are violated, then inconsistencies and feature interactions might occur and we cannot guarantee re-usability of test cases.)

In [15], we defined a theory for the evaluation of test cases with respect to a formal specification. The specification language CSP-CASL [22] allows to formalize features in a combined algebraic / process algebraic notation. As CSP-CASL has a loose semantics, a specification can be abstract, i.e., there may be non-determinate choices and open design decisions which are to be settled in a later design stage. In a formal systems development process, an abstract specification can be refined to a concrete implementation, where all design decisions have been fixed and which has a deterministic behaviour. In our approach, we can build test suites for any level of abstraction in this process. It is possible that test cases are constructed either from the specification or independently from it. Therefore, it is possible to structure a test suite according to the features of the SUT. Each test case checks the correct implementation of a certain feature according to a particular specification. The specification determines the alphabet of the test suite, and the expected result of each test case. The expected result is coded in a colouring scheme of test cases. If a test case is constructed which checks for the presence of a required feature (according to the specification), we define its colour to be green. If a test case checks for the absence of some unwanted behaviour, we say that it has the colour red. If the specification does neither require nor disallow the behaviour tested by the test case, i.e., if an SUT may or may not implement this behaviour, the colour of the test case is defined to be yellow. During the execution of a test on a particular SUT, the *verdict* is determined by comparing the colour of the test case with the actual behaviour. A test *fails*, if the colour of the test case is green but the SUT does not exhibit this behaviour, or if the colour is red but the behaviour can be observed in the SUT. The execution of a yellow test case yields an inconclusive verdict. Otherwise, the test passes. The validation triangle in Figure 1 shows an overview to our testing approach.

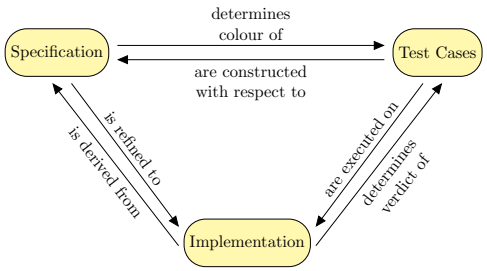

Figure 1. Validation triangle

In this paper, we investigate the extension of specifications and the re-use of test cases for software product lines. That is, we are not concerned with refinement relations between specification and implementation in a "vertical" systems development process, but with enhancement relations between different specifications in a "horizontal" process. We show that with a suitable enhancement concept, it is possible to re-use test cases from basic products for advanced products: The colour of a test case with respect to the basic specification is the same as the colour of this test case with respect to an enhanced specification. Therefore, test cases which have been designed for basic features can be re-used whenever a more advanced product is conceived which includes these features. We give semantical and syntactical conditions for the enhancement relation between specifications, show how to prove this property, and illustrate our ideas with a simple example of a common embedded device. Moreover, we report on a prototypical test execution framework for software product lines which implements our approach.

Related work There is a large body of literature on the automatic generation of tests from formal specifications (which is *not* the topic of our paper), see e.g. [9, 25]. In [8, 7], testing from label transition systems is studied, while in [5, 6] test generation methods for OO software are presented. All these approaches assume a finalised model or specification, whereas we are concerned with specifications which are being enhanced and combined from separate features.

The concept of a software product lines (SPL) was introduced in the late 1990's (see, e.g., [11, 14]), and extensively studied subsequently in [10, 20], with annual conferences and a huge body of engineering literature in [4, 2]. Testing for SPLs was investigated in [21, 17] and others; the main focus of these papers is the informal or formal derivation of test cases from requirement and feature models. Formal methods for SPLs have been studied in [16] and others; whereas these papers are mainly concerned with verification, we consider the combination and evaluation of test suites with respect to a formal specification.

Outline In Section 2 we give an overview of the specification language CSP-CASL and the test evaluation theory from [15]. In Section 3 we introduce our example of a remote control unit product line, and prove the enhancement property between different elements of this product line. Subsequently, in Section 4 we derive results about the re-use of test cases for different elements of a product line. In Section 5 we outline the implementation of a prototype testing framework which supports our approach.

2. Testing from CSP-CASL

In this section we describe the specification language CSP-CASL and the test evaluation framework from [15].

2.1 A short introduction to CSP-CASL

CSP-CASL [22] is a formal specification language that integrates the process algebra CSP [13, 24] with the algebraic specification language CASL [18]. The idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types specified in CASL. CSP-CASL includes all CSP standard operators. Also the full CASL language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting. Furthermore, the various CASL structuring constructs are included, where the structured **free** construct adds the possibility to specify data types with initial semantics. CSP-CASL specifications can be organized in libraries. This allows to specify a complex system in a modular way. See [12] for an industrial case study using CSP-CASL.

Syntactically, a CSP-CASL specification with name *Sp* consists of a data part D and a process part P. The data part is a structured CASL specification. The process part is written in CSP, where CASL terms are used as communications. CASL sorts denote sets of communications, relational renaming is described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions. The concrete syntax of a CSP-CASL specification is

ccspec *Sp* = **data** D **process** P **end**

See Section 3 for concrete instances of such a scheme. We often write shortly $Sp = (D, P)$.

Semantically, a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part D gives rise to one process denotation. The definition of the language CSP-CASL is generic in the choice of a specific CSP semantics, i.e., all denotational CSP models mentioned in [24] are possible parameters. We briefly sketch the semantic construction of a CSP-CASL specification, see [22] for details. The semantics of CSP-CASL is defined in a two-step approach. Given a CSP-CASL (D, P) specification:

- *first step*: for each CASL model M of D a CSP process $P'(\mathcal{A})$ is constructed. To this end, we define for each model M, which might include partial functions, an equivalent model $total(M)$, in which partial functions are totalized. $total(M)$ gives rise to an alphabet of communications $\mathcal{A} = A(total(M))$. In order to deal with CSP binding, variable evaluations $\nu : X \to total(M)$ are introduced. With these notations the process $P'(\mathcal{A}) := [\![P]\!]_{empty}$ is defined where *empty* denotes the empty evaluation and $[\![\_]\!]$ describes the evaluation according to CASL.
- *second step*: a denotational CSP semantics is applied for every model M. This translates a process $P'(\mathcal{A})$ into its denotation d_M in the semantic domain of the chosen CSP model.

Given a CSP model $\mathcal{D}$, the meaning of a CSP-CASL specification $Sp = (D, P)$ is a family of process denotations

$$(d_M)_{M \in \mathbf{Mod}(D)}.$$

In other words, every model M of the data part gives rise to one denotation in the domain $\mathcal{D}(\mathcal{A})$ of the CSP model $\mathcal{D}$ built relatively to the alphabet $\mathcal{A} = A(total(M))$.

The data part D of a CSP-CASL specification $Sp = (D, P)$ declares a subsorted signature $\Sigma = (S, TF, PF, P, \leq)$ which consists of a set of sort symbols S, a set of total functions symbols TF, a set of partial function symbols PF, a set of predicate symbols P, and a reflexive and transitive subsort relation $\leq \subseteq S \times S$ – see [18] for details.

In the following, we formally define how to extend a signature by new symbols to a larger one:

DEFINITION 2.1 *We say that a signature $\Sigma = (S, TF, PF, P, \leq)$ is **embedded into** a signature $\Sigma' = (S', TF', PF', P', \leq')$ if $S \subseteq S'$, $TF \subseteq TF'$, $PF \subseteq PF'$, $P \subseteq P'$, and the following conditions regarding the subsorting hold:*

preservation and reflection $\leq = \leq' \cap (S \times S)$.

weak non-extension *For all sorts $s_1, s_2 \in S$ and $u' \in S'$: if $s_1 \neq s_2$ and $s_1 \leq' u' \wedge s_2 \leq' u'$ then $u' \in S$.*

Weak non-extension means: whenever two different sorts, say s_1 and s_2 have a common supersort, say u', in the extended signature Σ', then u' must already be a sort of Σ. Thanks to the preservation and reflection property, this implies that s_1 and s_2 have u' as common supersort in Σ. We write $\iota : \Sigma \to \Sigma'$ for the induced map from Σ to Σ', where $\iota(s) = s, \iota(f) = f, \iota(p) = p$ for all sort symbols $s \in S$, function symbols $f \in TF \cup PF$ and predicate symbols $p \in P$.

In the following we re-call some relevant standard notations from algebraic specification: A Σ-model M gives an interpretation for all symbols, i.e. M_s a set for $s \in S$, M_f is a total function for $f \in TF$, M_f is a partial function for $f \in PF$, and M_p is a predicate for $p \in P$. Let D be a CASL specification, let Σ be its signature. We write $\mathbf{Mod}(D)$ for the class of all Σ-models that fulfil the axioms stated in D, see [18] for details. Let $\Sigma = (S, TF, PF, P, \leq)$ be embedded into $\Sigma' = (S', TF', PF', P', \leq')$. Then every Σ'-model M' defines a Σ-model $M' \mid_\iota$, the so-called reduct, which

simply forgets about the new symbols. Formally, we define $(M' \mid_\iota)_s = M'(\iota(s)) = M'_s$, $(M' \mid_\iota)_f = M'(\iota(s)) = M'_f$, and $(M' \mid_\iota)_p = M'(\iota(s)) = M'_p$ for all sort symbols $s \in S$, function symbols $f \in TF \cup PF$ and predicate symbols $p \in P$. For a specification D' with signature $\Sigma' = (S', TF', PF', P', \leq')$ we define $\mathbf{Mod}(D') \mid_\iota = \{M' \mid_\iota \mid M' \in \mathbf{Mod}(D')\}$.

Based on this semantics, [22] defines a refinement notion $\rightsquigarrow_{\mathcal{D}}$ on CSP-CASL specifications which is parametrized over the underlying CSP model $\mathcal{D}$.

$$(d_M)_{M \in I} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in I'}$$
$$\text{iff}$$
$$I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'},$$

where $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen CSP model $\mathcal{D}$.

2.2 Testing from CSP-CASL

In [15], a theory for the evaluation of test cases with respect to CSP-CASL specifications has been developed. In summary, the main benefits of this theory are as follows.

- Separation of test case construction from specification and implementation: A test case only refers to the signature of a specification. This allows to start the development of test suites as soon as an initial, abstract version of the specification is available, in parallel to the development of the actual implementation.
- Separation of the test oracle and the test evaluation problem: Test cases are constructed with respect to an (abstract) specification and executed on a (concrete) implementation. The specification determines the expected result of a test case, and the implementation the verdict. Therefore, the intrinsically hard test oracle problem can be solved before the actual execution, whereas test evaluation can be done online.
- Positive and negative test cases: The intention of a test case with respect to a specification is coded by a colouring scheme. It is possible to construct test cases for the intended behaviour (colour *green*) as well as for unwanted behaviour (colour *red*) of an implementation.
- Three-valued evaluation scheme for test cases: The colour of a test case as determined by a specification is either *green*, *red* or *yellow*; the test verdict is either *pass*, *fail* or *inconclusive*. If the colour of a test is determined to be yellow, this indicates that the respective behaviour is neither required nor disallowed by the specification. The test result is obtained by comparing intended and actual behaviour of an SUT. Yellow test cases lead to inconclusive test verdicts, indicating that the specification is not complete in this point.

Formally, a test case is just a CSP-CASL process in the same signature as the specification, which may additionally use first-order variables $x \in X$ ranging over communications. A *linear* test case is a process which can be written as $T = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow Stop$. Each test case validates the presence or absence of some feature described in the specification.

The *colour* of a test case T with respect to a CSP-CASL specification (D, P) is defined as follows.

- $colour_{Sp}(T) = green$ iff for all models $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \rightarrow M$ it holds that:
  1. $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{empty})$ and
  2. for all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_\nu)$ and for all $1 \leq i \leq n$ it holds that: $(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \notin failures([\![P]\!]_{empty})$
- $colour_{Sp}(T) = red$ iff for all models $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \rightarrow M$ it holds that: $traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{empty})$
- $colour_{Sp}(T) = yellow$ otherwise.

Intuitively, colour green means that for any data model all traces of the test case are possible system runs, whose execution can't be refused. A test case is red, if for each model some of its traces can not be executed by the system. A test case is yellow, if the execution is possible in some model and not possible or refused in some other model.

The execution of a test process is done with respect to a particular SUT. The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test process. In [15], we define the notions of *point of control and observation (PCO)*, which is a mapping between primitive events of an SUT and CASL-terms of the test case. A PCO $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$ of an SUT consists of: an alphabet $\mathcal{A}$ of primitive events, a mapping $\|...\| : \mathcal{A} \longrightarrow T_\Sigma(X)$ which returns for each $a \in \mathcal{A}$ a term over Σ, and a direction $D : \mathcal{A} \longrightarrow \{ts2sut, sut2ts\}$. *ts2sut* stands for signals which are sent from the testing system to the SUT, and *sut2ts* stands for signals which are sent in the other direction.

We say that a test case T is *executable* at a PCO $\mathcal{P}$ with respect to a specification *Sp*, if the mapping covers all CASL terms occuring in the test and is injective. We also define the *execution* of a test case on an SUT at a PCO. The test verdict is obtained by comparison of the actual behaviour of the SUT and the trace of the test case, where the noncompliance to a green test case or compliance to a red one leads to test failure, and the compliance to a green test case or noncompliance to a red one leads to passing the test.

3 Remote Control Unit Product Line

In this section we develop an example from the embedded systems domain: a product line of infrared remote control units, as they are used for home appliances such as TV, VCR, DVD-Player etc. We give abstract and concrete specifications of various elements of this product line in CSP-CASL and show how feature enhancement can be obtained.

On an abstract level, a remote control unit *RCU* can be described as follows: there are a number of *buttons* which can be pressed (one at a time), and a light emitting diode (*LED*) which is capable of sending *signals* (bitvectors of a certain length). The *RCU* has an internal table which signal correspond to which button. Whenever a button is pressed, it sends a corresponding signal via the LED. Such an Abstract Remote Control Unit can be specified in CSP-CASL as follows:

ccspec ABSRCU =
data
  sort $Button, Signal$
  op *codeOf* : $Button \to Signal$;
process
  AbsRCU = $?x : Button \to$ *codeOf*(x)
    $\to$ *AbsRCU*
end

Basic remote control units as they were produced in the 1970's had e.g. 12 buttons (i.e., $b_0 \ldots b_9, b_{OnOff}, b_{Mute}$), where the corresponding signals are 16-bit key-codes (e.g., $0000.01010.0000001$ for b_1) – see Figure 2. There are various standards for remote controls; one of these defines that the first 4 bits identify the company ID, the next 5 bits represent the device ID (i.e. TV, DVD, etc.), while the last 7 bits identify which button was pressed. This can be specified in CSP-CASL as follows:

ccspec BRCU =
data
  sort $Button, Signal$
  ops $b_0, b_1, \ldots, b_9, b_{OnOff}, b_{Mute} : Button$;
  free type $Bit ::= 0 \mid 1$
  then LIST[**sort** Bit]
    then
      sort $Signal = \{l : List[Bit] \bullet \#l = 16\}$
      op *codeOf* : $Button \to Signal$;
        $prefix : List[Bit] = [0000] ++ [01010]$
      axioms
        codeOf$(b_0) = prefix ++ [0000000]$;
        ...
        codeOf$(b_9) = prefix ++ [0001001]$;
        codeOf$(b_{Mute}) = prefix ++ [0001111]$;
        codeOf$(b_{OnOff}) = prefix ++ [1111111]$;
        $\forall b : Button \bullet \exists l : List[Bit] \bullet$
        codeOf$(b) = prefix ++ l$
process
  BRCU = $?x : Button \to$ *codeOf*$(x) \to$ *BRCU*
end

Without proof, we mention that BRCU is a refinement of ABSRCU: ABSRCU $\rightsquigarrow_{\mathcal{D}}$ BRCU, i.e., the specification BRCU is a correct implementation of the specification ABSRCU.

Soon after the first generation, the market demanded more comfortable devices with more functionality and, thus, more buttons. Modern remote control units have about 50-200 buttons. For the example, we assume that in the Extended specification the buttons b_{volup} and b_{voldn} for controlling the volume and b_{chup} and b_{chdn} for zapping though channels were added, with appropriate key-codes. In CSP-CASL, such an extension can be specified by defining a sort *EButton* which is an extension (superset) of sort *Button*. Of course, in the extended specification, the domain of operation *codeOf* must be suitably extended. Here is the abstract version of an extended remote control unit:

ccspec ABSERCU =
data
  sorts $Button < EButton$; $Signal$
  op *codeOf* : $Button \to Signal$;
    codeOf : $EButton \to Signal$;
process
  AbsERCU = $?x : EButton \to$ *codeOf*(x)
    $\to$ *AbsERCU*
end

Since the process *AbsERCU* communicates in a richer data structure, ABSERCU is not a refinement of ABSRCU.

For a concrete implementation of the abstract extended specification, we use the supersorting and overloading features built into CASL. To this end, we import the data part of BRCU, named BRCUDATA, and define a supertype *EButton* of *Button*, which includes four new buttons. The function *codeOf* : *EButton* $\to$ *Signal* is in overloading relation with the function *codeOf* : *Button* $\to$ *Signal*. Therefore, the CASL semantics ensure that both functions yield the same result for elements of type *Button*.

ccspec ERCU =
data BRCUDATA **then**
  free type $EButton ::=$ **sort** $Button$
    $\mid b_{volup} \mid b_{voldn}$
    $\mid b_{chup} \mid b_{chdn}$
  op *codeOf* : $EButton \to Signal$
  axioms
    codeOf$(b_{volup}) = prefix ++ [0010000]$;
    codeOf$(b_{voldn}) = prefix ++ [0100000]$;
    codeOf$(b_{chup}) = prefix ++ [1000000]$;
    codeOf$(b_{chdn}) = prefix ++ [1100000]$;
process
  ERCU = $?x : EButton \to$ *codeOf*$(x) \to$ *ERCU*
end

Again, without proof, we mention that

ABSERCU$\rightsquigarrow_{\mathcal{D}}$ERCU. However, BRCU and ERCU are not in a refinement relation, again, as the process of ERCU uses enhanced data structures. If more and more functions are added to a device, buttons need to be reused. That is, some buttons have a main and alternate inscription and there is a special button b_{alt}; if this button is pressed the key-code of the subsequently pressed button changes according to the alternate inscription. Basically, the button b_{alt} serves as a modifier of the next button. This enhancement differs from the previous one, since it requires the device to distinguish between two states (whether the b_{alt} modifier button has been pressed or has not been pressed), and it enforces a modification in the process part of the specification. The following is an (abstract) specification of a *RCU* with Modifier enhancing the extended *RCU*. To this end, we import the data part of ERCU, named ERCUDATA.

ccspec MERCU =
data ERCUDATA **then**
  free type $MButton ::=$ **sort** $EButton \mid b_{alt}$
  sort $AltButton = \{x : MButton \bullet x = alt\}$
  op *codeOfAlt* : $EButton \to Signal$
process
  MERCU =
    $?x : EButton \to codeOf(x) \to MERCU$
    $\Box\ b_{Alt} \to ?x : EButton \to codeOfAlt(x)$
      $\to MERCU$
end

As the *codeOfAlt* is not in overloading relation with *codeOf*, after pressing the button b_{Alt} the remote control can send out different signals for the buttons pressed. The specification MERCU is abstract in so-far, as the functionality of *codeOfAlt* is not further specified. In order to demonstrate the integration of features in a software product line development, we show how to reuse specification modules. A universal remote control is a device which can be used for *TV*, *VCR*, and *DVD* players. For this purpose, it has a button *mode*, which allows the user to cycle through the three operation modes (*TV*, *VCR*, *DVD*). The specification URCU builds onto the specification ERCU, as well as on similar built specifications ERCUDDVD and ERCUVCR.

ccspec URCU =
data { ERCUDATA **and** ERCUDATADVD
  and ERCUDATAVCR }
  then sort $NewButton$
    op $mode : NewButton$
process
  let TV $= ?x : Button \to codeOf(x) \to$ TV
    $\Box\ mode \to$ DVD
  DVD$= ?x : Button \to codeOfDVD(x) \to$ DVD
    $\Box\ mode \to$ VCR
  VCR$= ?x : Button \to codeOfVR(x) \to$ VCR
    $\Box\ mode \to$ TV
  in TV **end**

Figure 2. Basic Remote Control Unit (BRCU) – Screenshot from our Java implementation

3.1 Theory of enhancement

Analyzing the step from ABSRCU to ABSERCU, we can observe that all models of the data part of ABSRCU can be extended to models of the data part of ABSERCU. In algebraic specification this property is known as *conservative extension*, which is defined more precisely as

$$\mathbf{Mod}(D) = \mathbf{Mod}(D')|_\iota$$

where D and D' are CASL specifications with signatures Σ and Σ', respectively, and Σ is embedded in to Σ'.

Extensions with new symbols not necessarily are conservative. For an example, consider the following specifications BAS and EXT, where the new symbol c in EXT imposes a constraint on the symbols a and b inherited from BAS. Thus, models with $M(a) \neq M(b)$ of BAS are not included in $\mathbf{Mod}(\text{EXT})|_\iota$ and EXT is not a conservative extension of BAS.

spec BAS =
 sort S
 op $a, b : S$
end

spec EXT =
 sort $S < T$
 ops $a, b : S;\ c : T$
 axioms $c = a;\ c = b$
end

[23] compiles a comprehensive set of proof rules to establish that one specification conservatively extends another. For instance, the extension by the CASL construct 'operation definition' is conservative. In the following, we will use some of these rules in order to establish enhancement relations.

In the semantical construction of CSP-CASL, signature embeddings lead to alphabet embeddings:

LEMMA 3.1 *Let the signature Σ be embedded into Σ'. Let M' be a Σ' model. Then there exists an injection $\alpha : A(\beta(M'|_\iota)) \to A(\beta(M'))$ from the alphabet of the reduct of M' to the alphabet of M'.*

In analogy to the reduct on data, we define now a reduct on process denotations:

DEFINITION 3.2 *Let A and A' be sets such that there exists an embedding $\alpha : A \to A'$. Let, as usual in* CSP, *$\checkmark$ denote the symbol for successful termination. α extends in a canonical way to an injections $\alpha : A \cup \{\checkmark\} \to A' \cup \{\checkmark\}$ with $\alpha(\checkmark) = \checkmark$ and, by point-wise application, to a map on strings $\alpha : A^{*\checkmark} \to A'^{*\checkmark}$.*

1. *Let $T' \subseteq A'^{*\checkmark}$ be a set of traces. Then we define its reduct along α as $T'|_\alpha = \{t \in A^{*\checkmark} \mid \alpha(t) \in T'\}$.*
2. *Let $F' \subseteq A'^{*\checkmark} \times \mathcal{P}(A')$ be a set of failures. Then we define its reduct along α as $F'|_\alpha = \{(s,X) \in A^{*\checkmark} \times \mathcal{P}(A) \mid \exists (s',X') \in F' \bullet \alpha(t) = t' \wedge \alpha(X) = X' \cap \alpha(A)\}$.*

Note that our definitions subtly differ from the concept of eager abstraction and lazy abstraction as discussed, e.g., in [24]. Eager and lazy abstractions hide the new events in all traces – our approach, however, ignores traces that include new events.

CSP requires semantic denotations to be 'healthy', i.e., they have to fulfil certain constrains such as "a trace denotation must be non-empty and prefix-closed", see [24]. Process denotations over $\mathcal{T}$ and $\mathcal{F}$ remain healthy under reduct:

LEMMA 3.3 *With the same notations as above the following holds.*

1. *If T' is healthy over $\mathcal{T}$, the so is $T'|_\alpha$.*
2. *If (T', F') is healthy over $\mathcal{F}$, the so is $(T'|_\alpha, F'|_\alpha)$.*

Now we define the central notion of *enhancement* between specifications.

DEFINITION 3.4 *Let $Sp = (D, P)$ and $Sp' = (D', P')$ be* CSP-CASL *specifications, and let Σ and Σ' be the signatures of D and D', respectively. Let Σ and Σ' be the signatures of D and D', respectively. Let ι and α be the induced embeddings. Sp' is an* enhancement *of Sp, denoted by $Sp \gg Sp'$, if*

1. Σ is embedded into Σ',
2. $\mathbf{Mod}(D) = \mathbf{Mod}(D')|_\iota$, and
3. for all $M' \in \mathbf{Mod}(D')$ it holds that
   $traces([\![P]\!]_{empty}) = (traces([\![P']\!]_{empty}))\,|_\alpha$
   and
   $failures([\![P]\!]_{empty}) = (failures([\![P']\!]_{empty}))\,|_\alpha$

CSP-CASL enhancement guarantees preservation of behaviour up to the first communication that lies outside the original alphabet. This observation is captured in the following proof principle:

THEOREM 3.5 (EXTERNAL CHOICE ENHANCEMENT) *Let $Sp = (D, P = ?x : s \to P')$, let $Sp' = (D', P = ?x : s \to P' \;\Box ?y : t' \to Q')$, let Σ and Σ' be the signatures of D and D', respectively, let S be the set of sorts in Σ. If*

1. *Σ is embedded into Σ', $\mathbf{Mod}(D) = \mathbf{Mod}(D')|_\iota$, and*
2. *for all $u \in S$ it holds that $D' \models \forall x : u, y : t' \bullet x \neq y$,*

then $Sp \gg Sp'$.

Theorem 3.5 allows us to prove ERCU$\gg$MERCU. First, we have to adjust the process part of MERCU to the syntactic pattern stated in the theorem. To this end, we use the law $a \to R = ?x : \{a\} \to R[x/a]$ in order to transform $b_{Alt} \to ?y : EButton \to codeOfAlt(y) \to MERCU$ into $?z : AltButton \to ?y : EButton \to codeOfAlt(y) \to MERCU$. Concerning the data part MERCU is a conservative extension of ERCU, as all added symbols are new, and, if they relate to old ones, they follow a definitional extension pattern. Thanks to the CASL free type b_{alt} is different from all values of *EButton*. Thus, both conditions of Theorem 3.5 hold. In a similar way, we can establish with Theorem 3.5 that ERCU$\gg$URCU. Note, however, that $\neg$(ERCUDVD$\gg$URCU), as the DVD functionality is only available after pressing the mode button.

The enhancement from BRCU to ERCU makes use of overloading and added supersorts. To capture this technique by a characterization theorem, we introduce an extension operation, first on CASL signatures, then on CSP-CASL processes.

DEFINITION 3.6 *1. Given a mapping $extend : S \to S'$ on sort names, we define*

$extend(f) = f : extend(s_1) \times \cdots \times extend(s_k) \to extend(s)$ for a function symbol $f : s_1 \times \cdots \times s_k \to t$,

$extend(p) = p : extend(s_1) \times \cdots \times extend(s_k)$ for a predicate symbol $p : s_1 \times \cdots \times s_k$,

$extend(x : s) = x : extend(s)$ for a variable x of type s and

$extend(f(t_1, .., t_k)) =$
$extend(f)(extend(t_1), .., extend(t_k))$ for a CASL *term $f(t_1, \ldots, t_k)$,*

2. Σ is embedded into Σ' with a mapping $extend : S \to S'$ if Σ is embedded into Σ', $TF' = TF \cup extend(TF)$, $PF' = PF \cup extend(PF)$, $P' = TF \cup extend(P)$, and $\leq'$ is the minimal subsort relation with $\leq \subseteq \leq'$ and $(s, extend(s)) \in \leq'$.

The setting of Definition 3.6 ensures that any new function and predicate symbols in Σ' are in overloading relation with the old symbols of Σ.

For CSP-CASL processes, *extend* is the identity with the following two exceptions:

- $extend(t \to P) = extend(t) \to extend(P)$
- $extend(?x : s \to P) = ?x : extend(s) \to extend(P)$

THEOREM 3.7 (SUPERSORT ENHANCEMENT) *Let $Sp = (D, P)$, let $Sp' = (D', P')$, let Σ and Σ' be the signatures of D and D', respectively, let S and S' be the sets of sorts in Σ and Σ', respectively, let $extend : S \to S'$ be a mapping on sort names. If*

1. *Σ is embedded into Σ' with the mapping extend,*
2. ***Mod**$(D) =$ **Mod**$(D')|_\iota$, and*
3. *$P' = extend(P)$,*

then $Sp \gg Sp'$.

With Theorem 3.7 we can prove that BRCU$\gg$ERCU: To this end we define the map *extend* to be the identity on all sorts with the exception with the exception of $extend(Button) = EButton$. Clearly, the signatures are embedded with *extend*. As we define *codeOf* only for the new values, we have a conservative model extension. Obviously, *extend* maps the process of BRCU to the process of ERCU. Thus, all three conditions are true and therefore BRCU$\gg$ERCU. Similarly, Theorem 3.7 allows us to prove that ABSRCU$\gg$ABSERCU.

Presumably the demonstrated extension technique applies to more process operators, e.g., to sequential composition, external choice, internal choice, parallel.

Overall, the various relations between the specification of our product line of remote control units is summarized in Figure 3.

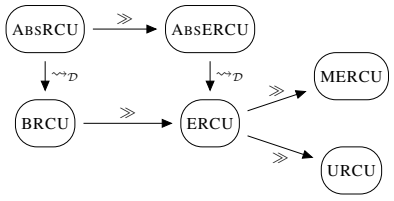

Figure 3. Remote Control SPL

4 Enhancement and Test Case Re-use

The previously defined enhancement relations allow for the re-use of results established w.r.t. the original specification: Test cases preserve their colouring and remain executable.

THEOREM 4.1 *Let Sp and Sp' be* CSP-CASL *specifications with $Sp \gg Sp'$. Let T be a test process over Sp. Then*

1. *$colour_{Sp}(T) = colour_{Sp'}(T)$.*
2. *Let $\mathcal{P}$ be a PCO.*

   If T is executable at $\mathcal{P}$ with respect to Sp, then T is executable at $\mathcal{P}$ with respect to Sp'.

In the following we use the first of these results in order to inherit test colouring along enhancements.

4.1 Remote Control Test cases

In this section we design some test cases for the *RCU* specifications and show the re-use of test cases as well as the preservation of colours described in the previous section. The first set of test cases is designed to test ABSRCU:

$$A_0 : u : Button \to codeOf(u) \to Stop$$
$$A_1 : u : Button \to v : Signal \to Stop$$
$$A_3 : u : Button \to w : Button \to Stop$$

Here, u, v and w are variable over the indicated sorts.

Thanks to the refinement and the enhancement results summarized in Figure 3, test cases T over ABSRCU are also test cases over all the other specifications. With respect to their colouring we obtain e.g. the following inheritance relations:

- $colour_{\text{ABSERCU}}(T) = colour_{\text{ABSRCU}}(T)$ thanks to enhancement.
- $colour_{\text{BRCU}}(T) = colour_{\text{ABSRCU}}(T)$ thanks to refinement.
- $colour_{\text{ERCU}}(T) = colour_{\text{ABSRCU}}(T)$, where we can either use the connection over BRCU or over ABSERCU.

This means for our three test cases A_0, A_1 and A_2 that their colour is the same in all specification mentioned in Figure 3, where their colouring can be determined by looking at ABSRCU only, i.e., the specification with the smallest number of axioms. For the colouring we obtain the following result respect to ABSRCU:

	A_0	A_1	A_2
AbsRCU	*G*	*Y*	*R*

A next set of test cases is designed to test BRCU:

$$T_0 : Stop$$
$$T_1 : b_1 \to Stop$$
$$T_2 : b_1 \to codeOf(b_1) \to b_6 \to codeOf(b_6) \to Stop$$
$$T_3 : b_1 \to b_6 \to Stop$$
$$T_4 : b_0 \to (prefix +\!+ [0000101]) \to Stop$$

The following table shows how these test process are coloured with respect to BRCU.

	T_0	T_1	T_2	T_3	T_4
BRCU	*G*	*G*	*G*	*R*	*R*

The empty observation T_0 is green with respect to all specifications. T_1 is green for *BRCU* as *BRCU* cannot refuse the event b_1 after the empty trace. The same holds for T_2, since *BRCU* cannot refuse the signal of b_1 after the event of b_1. T_3 consists of a sequence of two button presses and therefore is red for *BRCU*. T_4 however is red for *BRCU* due to a wrong signal event, i.e. $codeOf(b_0) \neq codeOf(b_5)$. Similarly to the result above, these test cases preserve these colours w.r.t. ERCU, MERCU and URCU.

In order to test the new features available in a the product line, new test cases have to be designed which use the new symbols. E.g., for ERCU the following test cases do this:

$T_5 : b_1 \rightarrow codeOf(b_1) \rightarrow b_{VolUp} \rightarrow codeOf(b_{VolUp}) \rightarrow Stop$
$T_6 : b_{ChUp} \rightarrow codeOf(b_{ChUp}) \rightarrow Stop$
$T_7 : b_{ChDn} \rightarrow codeOf(b_{ChDn}) \rightarrow b_1 \rightarrow Stop$
$T_8 : b_{ChUp} \rightarrow b_{VolDn} \rightarrow Stop$
$T_9 : b_{ChUp} \rightarrow codeOf(b_{VolUp}) \rightarrow Stop$

These test process are coloured with respect to ERCU in the following way:

	T_5	T_6	T_7	T_8	T_9
ERCU	*G*	*G*	*G*	*R*	*R*

These test cases preserve these colours w.r.t. MERCU and URCU.

5 Implementation

In this section we consider the evaluation of test cases w.r.t. CSP-CASL specifications from an implementation point of view. We also report on a prototypical framework for test execution. Figure 4 illustrates the basic flow of the testing process from a CSP-CASL specification, going into more details than Figure 1.

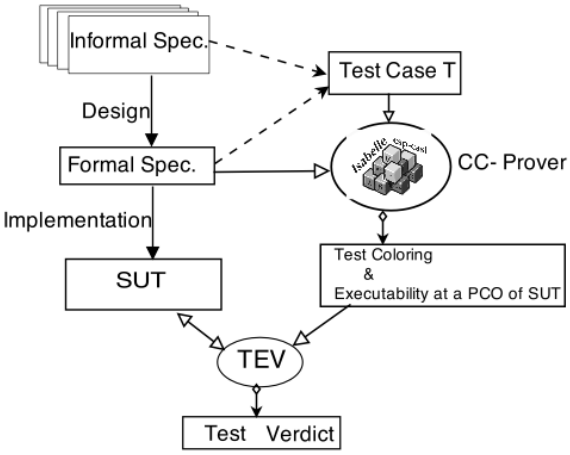

Figure 4. Testing from CSP-CASL

Our testing framework essentially consist of two parts which all have tool support:

1. We use CSP-CASL-prover [19] to verify the colour of a test case. To this end we use the syntactic characterization of the test colouring as defined in [15]. We also verify that a test case T is executable for the chosen PCO, see [15] for the definition.

2. Given a coloured test case and a particular SUT, our Test Execution and test Verdict program TEV, automatically runs a test against the SUT and automatically determines the test verdict.

In the following we discuss a prototype of TEV. Figure 5 shows a screen shot of the TEV front-end. In TEV we choose a coloured test case, e.g. $T2$ with colour green and an implementation, e.g. *J_BRCU*, which is a Java implementation of BRCU. TEV automatically executes and evaluates the test on the SUT – Figure 2 shows the animation of *J_BRCU* where `button6` has been pressed. In terms of executing tests against the SUT, we have implemented in Java the various version of the remote control discussed in section 3.

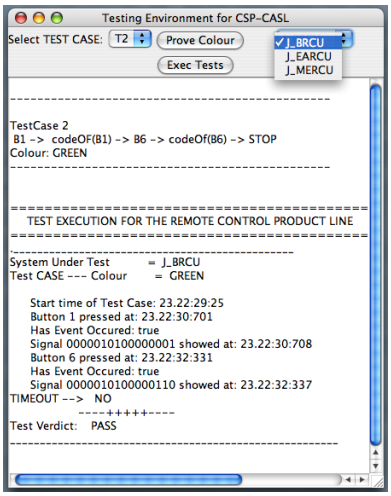

Figure 5. Testing Framework from CSP-CASL

In order to make the connection between the SUT and the testing system we use `abbot` [1], which is a package that enables to test Java AWT components. We setup a method that initialize the system under test and automatically finds the components of the system to be tested. In the case of the remote control unit it finds the different buttons. In order to execute test cases we instantiate robot-like objects, which automatically stimulate the SUT. For the execution of a test, we set a timeout of 2 secs as the period of time in which a signal is expected from the *RCU*. Depending on the colour of the test case and the response from the SUT, TEV determines automatically the test verdict. Figure 5 shows a test protocol of the execution of test case T_2 at *J_BRCU*. As the colour of T_2 w.r.t. BRCU is *green*, the expected signals

are correct and no timeout has occurred, the test verdict is *Pass*.

6 Summary and Future Work

In the paper, we have developed a framework for the enhancement of specifications and the re-use of test cases for software product lines. On the example of an embedded system, we developed a notion of specification enhancement which allows to add new user interfaces and extended behaviour to a system. These operations are fundamental for the feature-oriented engineering of a product line.

We proved that with our definition, the expected result of a test case is preserved; therefore, this notions allows to reuse test cases throughout a product line. We implemented a prototypical test execution framework, which automatically executes test cases and determines the test verdict on the fly. In the future, we plan to extend the capabilities of the testing framework, incorporate automated provers, and perform some larger case studies. On the theory side, there are other notions of enhancement which are not covered by our definition. For example, in object-oriented systems, re-use is by inheritance of signatures and methods: The enhanced version of a software product may inherit certain fields and classes, and redefine others. In order to re-use test suites for such a setting, general substitution operators for test cases will be necessary.

Acknowledgement The authors would like to thank Erwin R Catesbeiana (jr) for enhancing our knowledge on product lines.

References

[1] Abbot Java Gui Test framework. http://abbot.sourceforge.net.

[2] International Workshop on Software Product Line Testing 2007. http://www.biglever.com/split2007.

[3] Software Engineering Institute, Carnegie Mellon. http://www.sei.cmu.edu.

[4] Software Product Line Conference 2008. http://www.lero.ie/SPLC2008.

[5] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *EDCC-2*, London, UK, 1996. Springer-Verlag.

[6] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] E. Brinksma, W. Grieskamp, and J. Tretmans, editors. *Perspectives of Model-Based Testing*. IBFI, Schloss Dagstuhl, Germany, 2005.

[8] E. Brinksma and J. Tretmans. Testing transition systems: an annotated bibliography. In *Modeling and verification of parallel processes*. Springer, 2001.

[9] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[10] P. Clements and L. M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2001.

[11] P. C. Clements and N. Weiderman. Second international workshop on development and evolution of software architectures for product families. Technical Report CMU/SEI-98-SR-003, Carnegie Mellon University, 1998.

[12] A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment systems in CSP-CASL. In *Revised Selected Papers of WADT'04*, LNCS 3423. Springer, 2005.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] M. Jazayeri, A. Ran, and F. van der Linden. *Software architecture for product families: principles and practice*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[15] T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Specification-based testing for refinement. In M. Hinchey and T. Margaria, editors, *SEFM 2007*, pages 237–247. IEEE Computer Society, 2007.

[16] T. Kishi and N. Noda. Formal verification and software product lines. *Commun. ACM*, 49(12):73–77, 2006.

[17] J. D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, Software Engineering Institute, December 2001.

[18] P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960. Springer, 2004.

[19] L. O'Reilly, Y. Isobe, and M. Roggenbach. Integrating Theorem Proving for Processes and Data. In *CALCO-jnr 2007*. University of Bergen, 2008.

[20] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*, volume XXVI. Springer, 2005.

[21] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[22] M. Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354:42–71, 2006.

[23] M. Roggenbach and L. Schröder. Towards trustworthy specifications i: Consistency checks. In *WADT'01*, LNCS 2267. Springer, 2002.

[24] A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[25] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.