# Specification-based testing for refinement

Temesghen Kahsai* Markus Roggenbach* Bernd-Holger Schlingloff†

## Abstract

*In this paper, we present a theory for the evaluation of test cases with respect to formal specifications. In particular, we use the specification language* CSP-CASL *to define and evaluate black-box tests for reactive systems. Using loose semantics and three-valued test oracles, our approach is well-suited to deal with the refinement of specifications. In a structured development process of computational systems, abstract specifications are gradually refined into more concrete ones. With our approach, it is possible to develop test cases already from very abstract and basic specifications, and to reuse them later on in more refined systems.*

## 1. Introduction

Systematic testing is the most important quality assurance method in software and systems design. Testing can be done at all stages during the design, e.g. on unit-, integration-, and system level. System tests often are conceived as black-box-tests, where the inner structure of the system is hidden from the observer's view. In contrast to formal verification, black-box-testing is concerned with all parts of a computational system — software, middleware and hardware. The 'black box' view abstracts from the actual implementation details and considers the observable behaviour of the system only. The main purpose of testing is to determine whether or not a system under test (SUT) contains errors, where an error is a deviation of the actual from the intended behaviour of the SUT. If in a systematic test no errors are found, this can increase the confidence that the system is apt for its intended use.

Intentions are bound to individuals. For various reasons it is often necessary to establish testing results which are valid inter-subjectively. This requires that the intentions are denoted in an explicit, definite, and unambiguous specification. Within the design process of a system these intentions are developed in several steps. Consequently, especially the early specifications may leave room for later design decisions. In a specification, there may be properties which are intended, which are unwanted or which are left open. Several formal specification methods have been proposed. In this paper, we use the formal language CSP-CASL [16, 7] for the specification of a computational system. Likewise, the computational behaviour of a system can be denoted in a formal way, e.g. as a set of sequences of input/output events which occur at a certain point of control and observation (PCO). Such sequences are called *traces* of the SUT. Testing then becomes the task of comparing specified and traceable behaviour of a computational system, i.e., checking whether all intended behaviour is realized by traces, no unintended behaviour can be observed, and other behaviour is neither forced on the system nor inhibited. In this paper, we elaborate this idea of specification-based testing for distributed computer applications. We define a framework for test execution and evaluation based on a black-box SUT and formal CSP-CASL-specification.

Large computational systems often are developed in an incremental fashion, starting with an initial loose specification which leaves many design decisions open. This initial specification then is stepwise refined into more concrete artefacts; the last step of refinement is to transform a low-level specification into executable code which is ported to the SUT. It is generally agreed that the cost of error correction increases exponentially during the product life cycle. Hence, it is advisory that the testing process and the design of test cases starts as early as possible in the systems design. One of the main benefits of our approach is that the construction of test cases can start already in the very beginning, as soon as the first loose specifications are written. Our approach ensures that test cases which are designed at an early stage can be used without modification for the test of a later development stage. Hence, test suites can be developed in parallel with the SUT, which reduces the overall development time and helps to avoid ambiguities and specification errors.

Our work builds on previous work in specification-based testing, mainly in the area of LOTOS, see e.g. [9], [6], [3]. A first formal treatment of testing was given in [5]. The test oracle problem in such a setting was investigated in [12],

*University of Wales Swansea

†Humboldt University Berlin Fraunhofer FIRST

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

[13]. In contrast to these approaches, we are using a specification language with loose semantics which allows underspecification and refinement.

The language CSP dates back to 1985 [8]; an excellent reference is the book [17] (updated 2005). CASL was developed by the CoFI [1], [15], [2]. Tools for CASL are developed, e.g., in [10, 11, 14]. The combination CSP-CASL was introduced in [16] and used for specifying an electronic payment system in [7].

This paper is organized as follows: In section 2, we give a short overview of our specification language CSP-CASL. Then, we discuss reasonable expectations for specification based testing, where we use a simple calculator as running example. In section 4, we introduce the notion of a test process and the expected result of a test process with respect to a specification. In section 5, we describe test execution and evaluation as well as formal properties of test suites. Finally, we re-visit our calculator example and demonstrate our techniques.

## 2. CSP-CASL

CSP-CASL [16] is a comprehensive language which combines *processes* written in CSP [8, 17] with the specification of *data types* in CASL [2, 15]. The general 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, which are loosely specified in CASL. All standard CSP operators are available, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, communication over channels. Concerning CASL features, the full language is avaliable 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.

*Syntactically*, a CSP-CASL specification with name $N$ consists of a data part $Sp$, which is a structured CASL specification, an (optional) channel part $Ch$ to declare channels, which are typed according to the data part, and a process part $P$ written in CSP, within which 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:

**ccspec** $N$ = **data** $Sp$ **channel** $Ch$ **process** $P$ **end**

See Section 3 for concrete instances of such a scheme.

*Semantically*, a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part $Sp$ gives rise to one process denotation. The definition of the language CSP-CASL is generic in the choice of a specific CSP semantics. For example, all denotational CSP models mentioned in [17] are possible parameters, as is the newly defined model $\mathcal{R}$ [18].

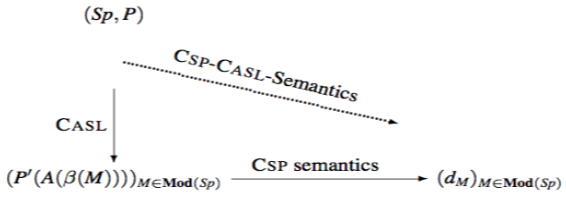


**Figure 1:** CSP-CASL semantics

The semantics of CSP-CASL is defined in a two-step approach[1], see Figure 1. Given a CSP-CASL $(Sp, P)$ specification, in the first step we construct for each model $M$ of $Sp$ a CSP process $P'(A(\beta(M)))$. To this end, we define for each model $M$, which might include partial functions, an equivalent model $\beta(M)$ in which partial functions are totalized. $\beta(M)$ gives rise to an alphabet of communications $A(\beta(M))$. In order to deal with CSP binding, we introduce variable evaluations $\nu : X \to \beta(M)$. With these notations we define the process $P'(A(\beta(M))) := [\![P]\!]_{\emptyset:\emptyset\to\beta(M)}$, where $\emptyset$ is the empty evaluation, i.e., $P$ has no free variables. In the second step we point-wise apply a denotational CSP semantics. This translates a process $P'(A(\beta(M)))$ into its denotation $d_M$ in the semantic domain of the chosen CSP model. See [16] for the details of the construction.

Given a denotational CSP model with domain $\mathcal{D}$, the semantic domain of CSP-CASL consists of families of process denotations $d_M \in \mathcal{D}$. Its elements are of the form $(d_M)_{M\in I}$ where $I$ is a class of algebras. As refinement $\rightsquigarrow_{\mathcal{D}}$ we define on these elements

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

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 $\mathcal{D}$. In the traces model $\mathcal{T}$, for instance, $T \sqsubseteq_{\mathcal{T}} T' :\Leftrightarrow T' \subseteq T$, where $T$ and $T'$ are prefixed closed sets of traces. The definitions of CSP refinements for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}, \mathcal{I}, \mathcal{U}\}$, c.f. [17], and also for the newly developed model $\mathcal{R}$ [18], which all are based on set inclusion, yield that CSP-CASL refinement is a preorder.

Concerning *data refinement*, we directly obtain the following characterisation:

$$\left.\begin{array}{c}\textbf{data } Sp \textbf{ process } P \\ \rightsquigarrow_{\mathcal{D}} \\ \textbf{data } Sp' \textbf{ process } P\end{array}\right\} \text{ if } \left\{\begin{array}{l}1.\ \Sigma(Sp) = \Sigma(Sp'), \\ 2.\ \textbf{Mod}(Sp') \subseteq \textbf{Mod}(Sp)\end{array}\right.$$

[1] We omit the syntactic encoding of channels into the data part.

The crucial point is that we fix both the signature of the data part and the process $P$. For *process refinement*, a similar characterisation is obvious:

$$\left.\begin{array}{c}\textbf{data } Sp \textbf{ process } P \\ \rightsquigarrow_{\mathcal{D}} \\ \textbf{data } Sp \textbf{ process } P'\end{array}\right\} \text{ if } \left\{\begin{array}{l}\text{for all } M \in \textbf{Mod}(Sp): \\ [\![[\![P]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} \\ [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}}\end{array}\right.$$

Here, $[\![\_]\!]_{\text{CSP}}$ is the evaluation according to the CSP denotational semantics. For this result, we need to fix $Sp$.

## 3 Relating formal specifications to tests

The fundamental question when dealing with testing based on formal specifications is the following: which are the intentions or properties formulated in the specification text that can be tested? In order to illustrate this question, we consider as an SUT a simple binary calculator, see Figure 2. It has two input buttons and can compute the addition function only. Whenever one of the buttons is pressed on the calculator, the integrated control circuit displays the corresponding digit in the display. After pressing a second button, the corresponding addition result is displayed and the calculator returns to its initial state. A major advantage of


**Figure 2:** Binary Calculator

specification-based design is the possibility of stepwise refinement. Starting from an initial, loose specification, more and more details can be added until the specification is effective, deterministic and executable. From this, the target code can be derived semi-automatically and in some cases even fully automatic.

Thus, the testing framework must be able to deal with incomplete and nondeterministic specifications. For the calculator example, in a first high-level specification we might want to abstract from the control flow and just specify the interfaces of the system.

**ccspec** BCALC0 =
**data**
**sort** $Number$
**ops** $0, 1 : Number$;
$\_\_ + \_\_ : Number \times Number \to ? Number$
**channel**
$Button, Display : Number$
**process**
$P_0 = (?x : Button \to P_0) \sqcap (!y : Display \to P_0)$
**end**

In this specification, 0 and 1 are constants of sort *Number*, and $+$ is a partial function from pairs of type *Number* to *Number*. The calculator receives values of type number on the channel *Button*, while it sends values of type number over the channel *Display*. The process $?x : Button \to P_0$ is willing to receive any value of type *Number* over the channel *Button*, stores this value of type *Number* in $x$, and behaves like $P_0$. This corresponds to a user input. The process $!y : Display \to P_0$ chooses an arbitrary value $y$ of type *Number*, sends this value over the channel *Display*, and behaves like $P_0$. This corresponds to the computed output of the calculator. The process $P_0$ as a whole repeatedly chooses one of the two above processes in a nondeterministic way, which corresponds to an arbitrary interleaving of inputs and outputs.

Even for such loosely specified systems we would like to be able to derive meaningful tests. For example, we could design test cases which are used for setting up the interface between testing system and SUT. The testing framework should be able to cope with such a situation.

A more refined specification could require that the pressing of buttons and the display of digits strictly alternates:

$$P_1 = ?x : Button \to !y : Display \to P_1$$

In the process $P_1$ each input is directly followed by some output. For such a specification, we would like to test that after each press of a button some digit is displayed.

An even more refined version requires that the first displayed digit is echoing the input, and the second displays the result of the computation:

$$\begin{aligned} P_2 = {} & ?x : Button \to Display!x \to ?y : Button \\ & \to Display!(x + y) \to P_2 \end{aligned}$$

The process $P_2$ has the following event sequence: The input of $x$ is followed by the display of the value $x$. Then the process reads $y$ and displays $x + y$. After that the process behaves like $P_2$ again. For testing this means that we can check if after input of $x$ the display shows $x$, and if after input of $x$ and $y$ the display shows the value of the term $x + y$.

Such refinement steps could occur, for example, when use cases which are derived from customer's wishes are integrated into the formal specification. Ideally, we would like to be able to re-use test cases on a more detailed level which have been designed for a more abstract level; since the refined specification is more precise than the abstract one, the outcome of testing should also be more precise. In particular, each test case developed for $P_1$ should be reusable for $P_2$.

In $P_2$ it is still left open what the value of $x + y$ shall be. We haven't yet specified the arithmetic properties of addition. Such situations of under-specifications occur, e.g., in

object-oriented design. Here, it is often the case that library functions are used whose exact functionality is specified at a later stage. In CSP-CASL, we can express functionality by suitable axioms:

**ccspec** BCALC3 =
**data**
  **sort** $Number$
  **ops** $0, 1 : Number$;
    $\_ + \_ : Number \times Number \to? Number$
  **axioms** $0 + 0 = 0$; $0 + 1 = 1$; $1 + 0 = 1$
**channel**
  *Button*, *Display* : *Number*
**process**
  $P_3 = ?x : Button \to Display!x$
    $\to ?y : Button \to Display!(x + y) \to P_3$
**end**

In BCALC3, we specify the results of the $+$ operator with three axioms. We deliberately under-specify the $+$ operator: there is no axiom for $1 + 1$. Consequently, BCALC3 has models with $1 + 1 = 0$, models with $1 + 1 = 1$ and models in which $1 + 1$ is undefined. This means that in this case the specification does not constrain the SUT in the result of the $+$ operator. Such a situation might for example arise when the functionality of border cases or exceptions is not constrained in the basic specification. E.g., in many programming languages the value of an integer variable in case of overflow is not defined. However, we want to design test cases which cover the normal, non-exceptional behaviour, and to re-use these test cases later on. With such a specification, we expect to be able to test whether the calculator behaves correctly, e.g., for the input of 0 and 1.

Taking the standard arithmetic CARDINAL from the CASL library of Basic Datatypes [15], we specify a one bit calculator where $1+1$ is seen as an arithmetic overflow and therefore is an undefined term.

**ccspec** BCALC4 =
**data**
  CARDINAL [**op** *WordLength* $= 1 :$ *Nat*]
  **with sort** *CARDINAL* $\mapsto$ *Number*
**channel**
  *Button*, *Display* : *Number*
**process**
  $P_4 = ?x : Button \to Display!x$
    $\to ?y : Button \to Display!(x + y) \to P_4$
**end**

For this specification all models of the data part are isomorphic, and in the process part there is no internal nondeterminism. Such a specification can be completely tested.

# 4 Test processes

In this section we define a classification of test processes based on the semantics of a CSP-CASL specification. Then we show that this classification can also be captured on the syntactical level.

## 4.1 Colouring test processes

In this subsection we formulate the notion of a test case and the colour of a test case with respect to a specification. Each test case reflects some intentions described in the specification. Intuitively, green test cases reflect *required* behaviour of the specification. Red test cases reflect *forbidden* behaviour of the specification. A test is coloured yellow if it depends on an *open design decision*, i.e., if the specification does neither require nor disallow the respective behaviour.

Let $(Sp, P)$ be a CSP-CASL specification such that $Sp$ is consistent, and let $X = (X_s)_{s \in S}$ be a variable system over the signature $\Sigma$ of the data part $Sp$. A *test case* $T$ is any CSP-CASL process in the signature of $Sp$ and the variable system $X$. The *colour* of a test case $T$ with respect to $(Sp, P)$ is a value $c \in \{red, yellow, green\}$, such that

- $colour(T) = green$ iff for all models $M \in \mathbf{Mod}(Sp)$ and all variable evaluations $\nu : X \to M$ it holds that:
  1. $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$ 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: $(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$
- $colour(T) = red$ iff for all models $M \in \mathbf{Mod}(Sp)$ and all variable evaluations $\nu : X \to M$ it holds that: $traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$
- $colour(T) = yellow$ otherwise.

In other words: a test case $T$ is green, if all models agree (1) that all its traces are possible system runs (2) whose execution can't be refused. A test case $T$ is red, if all models agree that not all of its traces are possible system runs and, finally, a test case $T$ is yellow, if the execution of some possible system run can also lead to failure, or the process $T$ has a trace which some models consider as a possible system run while others don't.

Our colouring has the following simple properties:

- The test process *Stop* which gives rise to the empty observation $\langle\rangle$ is always coloured green.
- With the specification $(Sp, Stop)$ all test cases different from *Stop* are coloured red.
- Test cases which differ only on termination information have the same colour.
- A CSP-CASL specification with an inconsistent data part $Sp$ does not reflect any intention, and, consequently, such a specification does not lead to any colouring of test cases.

- If $T$ and $T'$ are test cases such that $colour(T) = c$, $colour(T') = c'$, and $traces([\![T']\!]_\nu) \subseteq traces([\![T]\!]_\nu)$ for all models $M \in \mathbf{Mod}(Sp)$ and all variable evaluations $\nu : X \to M$, then $c = green$ implies $c' = green$ and $c' = red$ implies $c = red$.

Typical examples of open design decisions which lead to yellow test cases are the following:

**Internal nondeterminism** In the specification $Choice = (a \to b \to (P \mid\mid\mid Q)) \sqcap (b \to a \to (P \mid\mid\mid Q))$, the order of $a$ and $b$ is left open. Only the additional specification of a scheduling mechanism would enforce that, e.g., $a$ is executed at the first step. Consider the test case $T = a \to Stop$. As *Choice* has the failure $(\langle\rangle, \{a\})$, the colour of $T$ is yellow. At the first step of *Choice* neither the execution of $a$ nor the execution of $b$ can be guaranteed.

**Loose specification** In the same way, a heterogeneous model class of the data part can lead to a yellow test. Take for example the process (**if** $foo(1) = 2$ **then** $(continue \to Skip)$ **else** $(shutDown \to Skip)$). Here, the signal *continue* is sent in models where *foo* is, e.g., the successor function. In other models, *foo* might be the predecessor function and *shutDown* is sent. If the interpretation of *foo* is still an open design decision in the current specification, there are two correct implementations behaving differently. Consequently, the test case $(continue \to Skip)$ is coloured yellow.

The latter example illustrates also that the classification of a test process as green, red, or yellow is in general undecidable, as CASL includes full first order logic and arbitrary CASL predicates are allowed in case distinctions. This is called the *test oracle problem*, see e.g. [13][12].

### 4.2 Syntactic Characterisation

Using techniques originally developed in the context of full abstraction proofs for CSP [17], the above definition of colouring test processes has an equivalent syntactical characterisation for certain test processes. A test processes $T$ is called *linear* if it can be written as $T = t_1 \to \ldots \to t_n \to Stop$.

Concerning trace inclusion with respect to a linear test process $T$ of length $n \geq 0$, we define the following system of process equations

$$\begin{array}{lcl} check_T & = & ((P \parallel T)[[R_{s_1}]] \ldots [[R_{s_h}]] \\ & & |[A]| \, count(n)) \setminus s_1 \setminus \ldots \setminus s_n \\ count(n : Nat) & = & \textbf{if } n = 0 \textbf{ then } Ok \to Stop \textbf{ else } count(k-1) \end{array}$$

To make this a valid process part of a CSP-CASL specification, we take the original data part and extend it conservatively, i.e. without loosing any model of the data part. To this end we add a datatype *Nat* with the standard operations, a free type $A$ consisting only of the constant $a$, a free type $OK$ consisting only of the constant $Ok$, and for each sort $s_1 \ldots s_k$ in which $T$ could possibly communicate a renaming predicate $R_{s_i,A} : s_i \times A$ with the axiom $\forall x : s_i \bullet R(x, a)$.

We briefly sketch the idea behind this process definition: The synchronous parallel operator $\parallel$ forces $P$ and $T$ to agree on all communications. Should $P$ agree to execute the communications $t_1, \ldots, t_n$ of $T$ in exactly this order, this results in a sequence of $n$ communications. All these communications are renamed into $a$ via the predicates $R_{s_i,A}$. The process *count* communicates $Ok$ after the execution of $n$ $a's$. Hiding the communication $a$ makes only this $Ok$ visible.

Now, one can prove the following: Given a model $M \in \mathbf{Mod}(Sp)$ and a variable evaluation $\nu : X \to M$, then

$$\begin{array}{c} traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset \to \beta(M)}) \\ \Leftrightarrow \\ traces([\![check_T]\!]_\nu) = traces([\![Ok \to Stop]\!]_\nu) \end{array}$$

Thus, a test process is red, iff

$$\forall x_1 : s_1, \ldots, x_l : s_n \bullet check_T \neq_{\mathcal{T}} Ok \to Stop.$$

And for a test process to be green, it is necessary that

$$\forall x_1 : s_1, \ldots, x_l : s_n \bullet check_T =_{\mathcal{T}} Ok \to Stop.$$

Here $x_1, \ldots, x_l$ is the list of variables occurring in the terms.

A similar result can be obtained for the condition on the failures, which occurs in the colouring of a green test case.

## 5 Testing and Refinement

In the previous section, we defined the expected result of a test process with respect to a specification in terms of the colours green, red and yellow. In this section, we define how to execute a test case with respect to a particular SUT. Then, we study if the colouring of test cases is preserved under refinement. Finally, we show that the green test cases of a specification actually refine this very specification.

### 5.1 The overall test setting

Here, we define the execution of a test process with respect to a particular SUT, see Figure 3. The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test process.

A *point of control and observation* (PCO) $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$ of an SUT consists of

- an alphabet $\mathcal{A}$ of primitive events which can be communicated at this point,
- a mapping $\|...\| : \mathcal{A} \longrightarrow T_\Sigma(X)$ which returns for each $a \in \mathcal{A}$ a term (usually a constant) over $\Sigma$, and

- a direction $D : \mathcal{A} \longrightarrow \{ts2sut, sut2ts\}$.

*ts*2*sut* stands for signals which are sent from the testing system to the system under test, and *sut*2*ts* stands for signals which are sent in the other direction. In telecommunications, the mapping $\|...\|$ is called a *coding rule*. For the data type definition language ASN.1 (Abstract Syntax Notation One) [4] there are standardized coding rules for many frequently used PCOs.

We say that a test case $T$ is *executable* at a PCO $\mathcal{P}$ with respect to a specification $(Sp, P)$, if

- for each term $t$ occurring in $T$ there is exactly one $a_t \in \mathcal{A}$ such that $a_t$ and $t$ are equal[2]. This requirement ensures that each term in the test case corresponds to some observable or controllable event in the SUT.

- for $a, b \in \mathcal{A}$, if $\|a\|$ equals $\|b\|$ then $a$ and $b$ are the same primitive event. This requirement ensures that different observations or control events represent different values.

For test execution, we consider the SUT to be a process over the alphabet $\mathcal{A}$, where the internal structure is hidden. Hence, the SUT can engage in communications at the PCO. Communications $a$ with $D(a) = sut2ts$ are initiated by the SUT and are matched by the testing system with the expected event from the test case. Communications $a$ with $D(a) = ts2sut$ are initiated by the testing system and cannot be refused by the SUT. However, if the SUT sends an event without a stimulus, the SUT deviates from the specified behaviour. If the SUT internally refuses some communication, this can only be observed by the fact that it doesn't answer, i.e., the testing system waits for some event *sut*2*ts*, but this event does not happen. Testing is concerned with safety properties only; thus we say that in such a case a *timeout* happens.

The *test verdict* of a test case is defined relatively to a particular CSP-CASL specification and a particular SUT. The verdict is either *pass*, *fail* or *inconclusive*. Intuitively, the verdict *pass* means that the test execution increases our confidence that the SUT is correct with respect to the specification. The verdict *fail* means that the test case exhibits a fault in the SUT, i.e., a violation of the intentions described in the specification. The verdict *inconclusive* means that the test execution neither increases nor destroys our confidence in the correctness of the SUT.

Let $T = (t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow Stop)$ be a linear test case. Assume $colour(T) = c$ with respect to a specification $(Sp, P)$. Assume further that $T$ is executable at a PCO $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$. The test verdict of the test case $T$ with colour $c$ at the PCO $\mathcal{P}$ relatively to an execution of the SUT is defined inductively as follows:

- If $n = 0$ the colour $c$ of the test case yields the test verdict as follows: if $c = green$ the test verdict is *pass*, if $c = red$ the test verdict is *fail*, if $c = yellow$ the test verdict is *inconclusive*.

- If $n > 0$, let $a$ be the primitive event with $\|a\|$ equals $t_1$. Assume that the colour $c$ is

  - *green*: If the direction $D(a) = sut2ts$ and we receive $a$, then we inductively determine the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow Stop)$.

    If the direction $D(a) = sut2ts$ and we receive some $b$ different from $a$ or if a timeout occurs, then the test verdict is *fail*.

    If the direction $D(a) = ts2tsut$ and we receive an event from the SUT within the timeout period, then the test verdict is *fail*.

    If the direction $D(a) = ts2sut$ and we do not receive an event during the timeout period, then we send $a$ to the SUT and obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow Stop)$.
  - *red*: If the direction $D(a) = sut2ts$ and we receive $a$ we obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow Stop)$.

    If the direction $D(a) = sut2ts$ and we receive some $b$ different from $a$ or if a timeout occurs, then the test verdict is *pass*.

    If the direction $D(a) = ts2tsut$ and we receive an event from the SUT within the timeout period, then the test verdict is *pass*.

    If the direction $D(a) = ts2sut$ and we do not receive an event during the timeout period, then we send $a$ to the SUT and obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow Stop)$.
  - *yellow*: the test verdict is *inconclusive*.

The verdict of a yellow test case is always *inconclusive* and does not require any execution of the SUT. Recall that a yellow test case reflects an open design decision. Consequently, such a test case can neither reveal a deviation from the intended behaviour, nor can it increase the confidence that the system is apt to its intended use. After taking this design decision, however, i.e. turning the property into

[2] Strictly speaking, this is the case if $a_t$ and $t$ are of the same sort and $Sp \models (\forall X \bullet \|a_t\| = t)$. Here, $X$ is a variable system including all variables of $\|a_t\|$ and $t$. Since in general equality of CASL terms is undecidable, in general it is undecidable if an arbitrary test case is executable with respect to a PCO. However, for all practical purposes equality is easily decidable.

an intended or a forbidden one, the colour of the test will change and we will obtain *pass* or *fail* as a verdict. In the next section we will discuss how the colouring of test cases changes under refinement.

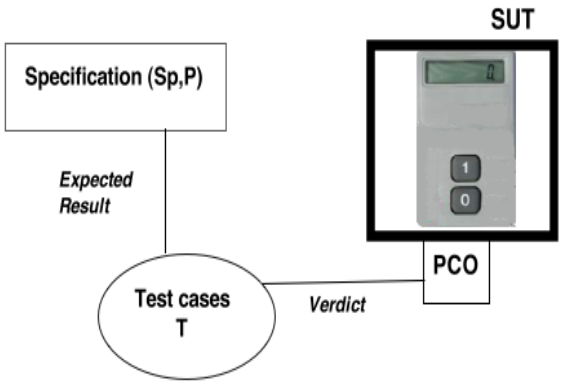


**Figure 3:** Overall formal testing framework

## 5.2 Properties of Test-Suites

Let $\leq$ be a binary relation over CSP-CASL specifications such that $(Sp, P) \leq (Sp', P')$ implies that the signature of $Sp$ is included in the signature of $Sp'$. We call such a relation to be $\leq$ *well-behaved*, if, given specifications $(Sp, P) \leq (Sp', P')$ with consistent data parts $Sp$ and $Sp'$ and a variable system $X$ over the signature of $Sp$, the following holds for any test process $T$ over $Sp$:

1. $colour(T) = green$ with respect to $(Sp, P)$ implies $colour(T) = green$ with respect to $(Sp', P')$, and
2. $colour(T) = red$ with respect to $(Sp, P)$ implies $colour(T) = red$ with respect to $(Sp', P')$.

Interpreting $\leq$ as a development step, this means: If a test case $T$ reflects a desired behavioural property in $(Sp, P)$, i.e. $colour(T) = green$, after a well-behaved development step from $(Sp, P)$ to $(Sp', P')$ this property remains a desired one and the colour of $T$ is green. If a test case reflects a forbidden behavioural property in $(Sp, P)$, i.e. $colour(T) = red$, after a well-behaved development step from $(Sp, P)$ to $(Sp', P')$ this property remains a forbidden one and the colour of $T$ is red. A well-behaved development step can change only the colour of a test case $T$ involving an open design decision, i.e. $colour(T) = yellow$. Figure 4 illustrates this effect in the context of the test verdict.

In the following, we study for various refinement relations if they are well-behaved.

### 5.2.1 Data refinement is well-behaved

Let $(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P)$ hold via data refinement, i.e. $\Sigma(Sp) = \Sigma(Sp')$ and $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp)$. Then $\rightsquigarrow_{\mathcal{D}}$ is

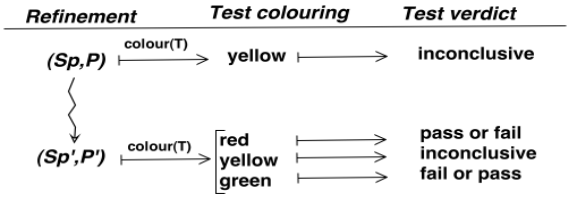


**Figure 4:** Test colouring and test verdict under well-behaved refinement

well-behaved independent of the choice of the CSP model $\mathcal{D}$. Especially, data refinement over the traces model $\mathcal{T}$, the failures/divergences model $\mathcal{N}$, the stable failures model $\mathcal{F}$, and the stable revivals model $\mathcal{R}$ [18] are well behaved.

Of this proof we consider only the case that the test process $T$ is green over $(Sp, P)$. Let $M'$ be model of $Sp'$. As $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp)$, we obtain: $M'$ is also a model of $Sp$. For the models of $Sp$, however, the conditions 1. and 2. for a green test case hold by assumption.

### 5.2.2 Model $\mathcal{T}$: process refinement is not well-behaved

As the CSP trace refinement does not guarantee the preservation of behaviour, it is to be expected that the CSP-CASL notion of process refinement based on $\mathcal{T}$ fails to be well-behaved. This is illustrated by the following counterexample:

| **ccspec** DOONEA = | **ccspec** DONOTHING = |
|---|---|
| **data sort** $s$ | **data sort** $s$ |
| **op** $a : s$ | **op** $a : s$ |
| **process** | **process** |
| $a \rightarrow Stop$ | $Stop$ |
| **end** | **end** |

As *Stop* refines any process in the CSP traces model $\mathcal{T}$, we have DOONEA $\rightsquigarrow_{\mathcal{T}}$ DONOTHING via process refinement. The process $T = a \rightarrow Stop$ has the colour green over DOONEA, however has the colour red over DONOTHING.

### 5.2.3 Models $\mathcal{N}$, $\mathcal{F}$ and $\mathcal{R}$: process refinement is well-behaved for divergence free processes

CSP-CASL process refinement based on CSP models $\mathcal{N}$, $\mathcal{F}$ and $\mathcal{R}$ is well-behaved, provided the processes involved are divergence-free.

Concerning $\mathcal{N}$ and $\mathcal{F}$ it is sufficient to prove this for $\mathcal{F}$ only, as failures divergences refinement and stable failures refinement are equivalent on divergence free processes. Again, we consider only the case of a green test process.

Let $(Sp, P) \rightsquigarrow_{\mathcal{F}} (Sp, P')$ hold via process refinement. Let the test process $T$ be green with respect to $(Sp, P)$. Let $M$ be a model, let $\nu$ be a variable evaluation. We prove by induction on the length $n$ of traces $t \in traces([\![T]\!]_\nu)$ that $t \in traces([\![P]\!]_{\emptyset:\emptyset \rightarrow \beta(M)})$. For $n = 0$ this is obviously the case. Let $t = \langle t_1, \ldots, t_n, t_{n+1} \rangle \in traces([\![P]\!]_{\emptyset:\emptyset \rightarrow \beta(M)})$. Then

$\langle t_1,\ldots,t_n\rangle \in traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ and thus by induction hypothesis also $\langle t_1,\ldots,t_n\rangle \in traces([\![P'']\!]_{\emptyset:\emptyset\to\beta(M)})$. Let us assume that $\langle t_1,\ldots,t_n,t_{n+1}\rangle \notin traces([\![P'']\!]_{\emptyset:\emptyset\to\beta(M)})$. As a divergence free process, $P''$ has the failure $(\langle t_1,\ldots,t_n\rangle,\emptyset)$. Thus, by healthiness of the stable failures domain, $P''$ has also failure $(\langle t_1,\ldots,t_n\rangle,\{t_{n+1}\})$ – which is a contradiction to $failures([\![P'']\!]_{\emptyset:\emptyset\to\beta(M)}) \subseteq failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}) \not\ni (\langle t_1,\ldots,t_n\rangle,\{t_{n+1}\})$. The proof of the 2nd condition is trivial.

The proof for $\mathcal{R}$ follows the same line of argument.

### 5.3 Green test cases form a refinement

Besides the question whether refinements are well behaved with respect to test case colouring, one can also ask the other way round: Is there a refinement between the specification and the test processes? Here, we have the relation: Given a green test $T$ over a CSP-CASL specification $(Sp,P)$. Then $(Sp,P) \rightsquigarrow_{\mathcal{T}} (Sp,T)$, i.e. every green test process is a CSP-CASL process refinement with respect to the traces model $\mathcal{T}$. Let $Green_{(Sp,P)}$ be the set of all green test processes with respect to $(Sp,P)$. Then $(Sp,P) \rightsquigarrow_{\mathcal{T}} (Sp,\bigsqcap Green_{(Sp,P)})$.

## 6 The calculator example revisted

In this section we re-consider the calculator example stated in Section 3 and establish the informal claims and promises made in that section. We prove refinement relations between the various specifications, construct a test suite, and determine the colour of its test cases with respect to the specification of Section 3. Finally, we show how the test cases can be implemented with a software simulation of the calculator.

### 6.1 Refinements

We first show by fixed point induction that BCALC0 refines to BCALC1 via process refinement within the stable failures model $\mathcal{F}$.

$$\begin{array}{ll} & (?x:Button \to P_0) \sqcap (!y:Display \to P_0) \\ =_{\mathcal{F}} & (?x:Button \to ((?x:Button \to P_0) \\ & \sqcap\ (!y:Display \to P_0))) \sqcap\ (!y:Display \to P_0) \\ \sqsubseteq_{\mathcal{F}} & ?x:Button \to ((?x:Button \to P_0) \\ & \sqcap\ (!y:Display \to P_0)) \\ \sqsubseteq_{\mathcal{F}} & ?x:Button \to !y:Display \to P_0 \end{array}$$

The proofs uses standard CSP laws: First, we unwind the recursion at the first occurrence of $P_0$. Then we leave out the second branch of the last internal choice. Finally, we select the second branch of the internal choice.

Next, we show that BCALC1 refines to BCALC2 via process refinement within the stable failures model $\mathcal{F}$.

$$\begin{array}{ll} & ?x:Button \to !y:Display \to P_1 \\ =_{\mathcal{F}} & ?x:Button \to !y:Display \\ & \to ?x':Button \to !y:Display \to P_1 \\ \sqsubseteq_{\mathcal{F}} & ?x:Button \to Display!x \to ?x':Button \\ & \to Display!(x+x') \to P_1 \\ =_{\mathcal{F}} & ?x:Button \to Display!x \to ?y:Button \\ & \to Display!(x+y) \to P_1 \end{array}$$

Again, we first unwind the recursion of $P_1$. As $P_1$ is independent of $x$ we can rename $x$ into $x'$. Chosing the specific values $x$ and $x+x'$ for the two occurences of $!y:Display$ is a refinement. Finally, renaming $x'$ into $y$ preserves the semantics of the process.

As all processes involved are divergence free, our proofs also carry over to the failure/divergences model $\mathcal{N}$.

Adding axioms to a signature without changing the process part always results in a data refinement. Consequently, BCALC2 refines to BCALC3 via data refinement. As the axioms stated in BCALC3 hold, e.g., for the natural numbers, the data part of BCALC3 is consistent.

The axioms for 0 and 1 stated in BCALC3 hold in the specification CARDINAL. As the process part is not changed, BCALC3 refines to BCALC4 via a data refinement.

### 6.2 Colouring of test cases

In this subsection we define some meaningful test cases for the calculator.

$T_0 : Stop$
$T_1 : Button!1 \to Stop$
$T_2 : Display!1 \to Stop$
$T_3 : Button!0 \to Button!1 \to Stop$
$T_4 : Button!1 \to Display!1 \to Stop$
$T_5 : Button!1 \to Display!0 \to Stop$
$T_6 : Button!0 \to Display!0 \to Button!1 \to Display!1 \to Stop$
$T_7 : Button!1 \to Display!1 \to Button!1 \to Display!0 \to Stop$

The following table shows how these test process are coloured with respect to the specifications of Section 3:

| | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | G | Y | Y | Y | Y | Y | Y | Y |
| $P_1$ | G | G | R | R | Y | Y | Y | Y |
| $P_2$ | G | G | R | R | G | Y | Y | Y |
| $P_3$ | G | G | R | R | G | R | G | Y |
| $P_4$ | G | G | R | R | G | R | G | R |

The empty observation $T_0$ is green with respect to all specifications. $T_1$ becomes green for $P_1$ as $P_1$ cannot refuse the event $Button!1$ after the empty trace. $T_2$ becomes red for $P_2$ as $Display!1$ is not an initial event of $P_2$. $T_6$ stays yellow for

$P_0$ to $P_2$, as the result of $0+1$ has yet not been specified. The axiom $0+1=1$ in $P_3$ turns this test case green. However, the result of $1+1$ is still not specified in $P_3$. Thus, $T_7$ is still yellow in $P_3$. Only for $P_4$ we know that $1+1$ is undefined. As the constant 1 is defined, however, $P_7$ becomes red over $P_4$. Note that – in accordance to our theoretical results – the colour of a green or red test case does not change under refinement.

We also give a sample proof for these colourings using the syntactical characterisations stated in Section 4.2. We show that the test case $T_6$ is coloured green with respect to $P_3$:

$$
\begin{array}{ll}
 & \mathit{check}_{\mathcal{T}} \\
= & ((\quad (?x : \mathit{Button} \rightarrow \mathit{Display}!x \rightarrow \\
 & \qquad ?y : \mathit{Button} \rightarrow \mathit{Display}!(x+y) \rightarrow P_3) \\
 & \quad \| \quad (\mathit{Button}!0 \rightarrow \mathit{Display}!0 \rightarrow \\
 & \qquad \mathit{Button}!1 \rightarrow \mathit{Display}!1 \rightarrow \mathit{Stop}) \\
 & )[[a/\mathit{Button}!0, \ldots, a/\mathit{Display}!1]] \,|[\, a \,]|\, \mathit{count}(4)) \setminus \{a\} \\
= & (\quad (\mathit{Button}!0 \rightarrow \mathit{Display}!0 \rightarrow \\
 & \qquad \mathit{Button}!1 \rightarrow \mathit{Display}!1 \rightarrow \mathit{Stop}) \\
 & [[a/\mathit{Button}!0, \ldots, a/\mathit{Display}!1]] \,|[\, a \,]|\, \mathit{count}(4)) \setminus \{a\} \\
= & (a \rightarrow a \rightarrow a \rightarrow a \rightarrow \mathit{Ok} \rightarrow \mathit{Stop}) \setminus \{a\} \\
= & \mathit{Ok} \rightarrow \mathit{Stop}
\end{array}
$$

Thus, the traces condition for green is fulfilled. As $P_3$ does not have any internal non-determinism, also the failures condition becomes true. Consequently, the colour of the test case $T_6$ is green.

### 6.3 Establishing a SUT and PCO

In order to demonstrate how our approach can be used for testing an implementation, we sketch a Java program which realizes the specification BCALC4. We give only the parts necessary to understand the communication and omit most of the control part.

```
import java.awt.*;
public class BCalc extends JFrame {
 ...
public BCalc() { initialize(); }
private JPanel getJContentPane()
{ ...
button0=new JButton();
button0.addActionListener
 (new ActionListener()
{public void actionPerformed
                    (ActionEvent e)
    { execute(0); }});
    pane1.add(button0);
    ...
    display = new JTextField(12);
    pane2.add(display);
    ... }
```

```
protected void execute(int value)
 { if (firstPress)
  { displayValue=value;
    firstPress=false;
    display.setText(""+displayValue); }
  else { if (displayValue==0 && value==0)
    { displayValue=0;
      display.setText(""+displayValue);
      firstPress=true; }
    else { ... } } }

public static void main(String[] args)
 { new BCalc(); } }
```

Upon startup, this program creates objects for the buttons and display. For each button there is a private method `actionPerformed` which is called by the Java runtime library whenever the button is pressed with the mouse. The program then uses the method `setText` inherited from `JTextField` to write the result to the screen.

For the point of control and observation of this SUT we use the alphabet of primitive events $\mathcal{A} = \{\mathit{button}_0, \mathit{button}_1, \mathit{display}_0, \mathit{display}_1\}$. We define $\|\mathit{button}_i\| = \mathit{Button}.i$, $\|\mathit{display}_i\| = \mathit{Display}.i$ for $i = 0, 1$. The direction of button events is *ts*2*sut*, while it is *sut*2*ts* for display events. The test cases $T_1$ to $T_7$ of Section 6.2 are executable at the above PCO with respect to all specifications of Section 3.

In order to execute the test cases the testing system must be able

- to invoke the appropriate method `actionPerformed` for each button event and
- to observe the respective display event whenever the method `setText` is called.

For the execution of the test processes, we define 2 secs as the period of time in which we expect a response from the SUT. Interesting test executions arise from the test cases $T_2$ and $T_3$. Here, we first show a test protocol of the execution of $T_2$ at our implementation of BCALC4 in Java:

```
Colour of the test: red
Start time of the test:       7:28:21:021
No event was recieved after 2 sec: timeout.
   Test verdict: pass.
```

As $T_2$ is red over BCALC4 and a timeout occurs, the test verdict is pass.

Concerning $T_3$ we obtain the following execution protocol:

```
Colour of the test: red
Start time of the test:       7:30:22:031
button0 was pressed at   time 7:30:24:036
display0 was received at time 7:30:25:236
no events were expected from the SUT
   Test verdict: pass.
```

Within the defined period of 2 secs we do not receive an event from the SUT. Thus, we press *button*0 and continue executing the test case. We then receive *display*0 from the SUT within 2 secs – as $D(button1) = ts2sut$, the test verdict is pass.

## 7 Summary and Future Work

In this paper, we have developed a theory for the evaluation of test cases with respect to formal specifications. The major innovations are:

- We separate the test oracle and the test evaluation problem by defining the expected result (*green*, *red* and *yellow*) and the verdict (*pass*, *fail* and *inconclusive*) of a test case.
- We use a three-valued colouring scheme for the expected result, which allows specification transformation and refinement, and
- We use a three-valued evaluation scheme for the verdict, which allows tests to be performed at all stages in a system's design.

We obtained some theoretical results on our approach and gave a practical example.

There are a number of questions which arise from this work and could not be addressed in the present paper. We already mentioned the question of test generation and test coverage. Although our framework suggests an algorithm for online test execution and monitoring based on CSP-CASL-provers, a practical implementation of such an algorithm is still missing. On the theoretical side, there are other refinement and simulation notions which could be considered and analysed for well-behavedness. Lastly, we want to consider relations between specifications which also allow to change the signature of the data part. Such *enhancement* relations would enable us to re-use test suites for new products in a product line, where each product specification enhances the previous one.

**Acknowledgement** The authors would like to thank Erwin R Catesbeiana (jr) for pointing out the possibility of on-the-fly evaluation of test verdicts.

## References

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, Sept. 2002.

[2] M. Bidoit and P. D. Mosses. CASL *User Manual*. LNCS 2900. Springer, 2004.

[3] E. Brinksma, G. Scollo, and C. Steenbergen. Lotos Specifications, their Implementations, and their Tests. In *Protocol Specification, Testing and Verification*, pages 349–360. Elsevier, 1987.

[4] O. Dubuisson. *ASN.1 communication between heterogeneous systems*. Morgan Kaufmann, 2000.

[5] M.-C. Gaudel. Testing can be formal, too. In *CAAP/FASE*, LNCS 915. Springer, 1995.

[6] M.-C. Gaudel and P. R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 1999.

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

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

[9] ISO 8807. Lotos — a formal description technique based on the temporal ordering of observational behaviour, 1989.

[10] C. Lüth, M. Roggenbach, and L. Schröder. CCC —the CASL Consistency Checker. In *WADT 2004*, LNCS 3423. Springer, 2005.

[11] K. Lüttich and T. Mossakowski. Reasoning support for CASL with automated theorem proving systems. In *WADT 2006*, LNCS 4409, 2007.

[12] P. D. L. Machado. On oracles for interpreting test results against algebraic specifications. In *AMAST'99*, LNCS 1548. Springer, 1999.

[13] P. D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University Edinburgh, 2000.

[14] T. Mossakowski, C. Maeder, and K. Lüttich. The heterogeneous tool set. In *TACAS 2007*, LNCS 4424, 2007.

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

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

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

[18] B. Roscoe. Revivals,stuckness and responsiveness, 2005. unpublished draft.