Testing of component-based systems

Bilal Kanso
École Centrale Paris
Grande Voie des Vignes
F-92295 Châtenay-Malabry
bilal.kanso@ecp.fr

Marc Aiguier
École Centrale Paris
Grande Voie des Vignes
F-92295 Châtenay-Malabry
marc.aiguier@ecp.fr

Fréderic Boulanger
Supelec E3S
3 rue Joliot-Curie
F-91192 Gif-sur-Yvette cedex
frederic.boulanger@supelec.fr

Christophe Gaston
CEA LIST Saclay
F-91191 Gif-sur-Yvette cedex
christophe.gaston@cea.fr

***Abstract*—In this paper, we pursue our works on generic modeling and testing of component-based systems. Here, we extend our conformance testing theory to the testing of component-based systems. We first show that testing a global system can be done by testing its components thanks to the projection of global behaviors onto local ones. Secondly, based on our projection techniques, we define a framework to build adequate test purposes automatically for testing components in the context of the global system where they are plugged in. The basic idea is to identify from any trace *tr* of the global system, the trace of any component involved in *tr*. Those projected traces can be then seen as test cases that should be tested on individual components.**

Keywords: Component-based system, Conformance testing, Compositional testing, Testing in context, Projection, Test purpose.

INTRODUCTION

In the last decades, the component-based software approach [1], [14] has emerged due to the great advantages it offers: modularity, re-usability, etc. Components are then designed, developed and validated in order to be widely used, while complex software systems are described recursively as interconnections of those components. Hence, each sub-system (or component) can be either a complex system itself or a simple component, elementary enough to be handled without further decomposition. Composition is used for fitting different components together and then defining larger systems. Such a composition is defined by operations which take components as well as the nature of their interactions to provide a description of a new and more complex component or system.

In [2], we proposed a formal framework for modeling basic components viewed as abstract state-based systems. Components were then modeled as coalgebras over **sets**-endofunctor with monads [3], [4] following *Barbosa*'s component definition [5], [6]. This definition allowed us to unify in a same framework a large family of state-based formalisms such as Mealy automata [7], Labeled Transition Systems [8], etc. Larger systems are then built by integrating components using integration operators defined by composition of two basic ones: Cartesian product and feedback. In [2], we showed that most standard integration operators such as sequential or synchronous product are subsumed by our generic definition of integration operators. Based on this framework, a conformance testing theory has been defined in [2].

The "plug and play" nature of component-based system design leads naturally to build always bigger systems whose correctness happens to be increasingly difficult to assert. This is due to the fact that analyzing big systems generates state and time explosion problems, but it may also be caused by the system architecture (e.g. distributed system) which may complicate the ability to instrument the system in order to observe behaviors to be analyzed. Even more, if a "faulty" behavior is observed in such a system, the system size is a problem to identify the fault cause at the debugging phase. All these reasons call to find ways to make system validation modular. Such methods enable to analyze a system, subsystems per subsystems, in a modular way, rather than "as a whole". Such analyzed systems are smaller (less prone to generate explosion problems), more observable and controllable (their behaviors are easier to cover), and debugging is greatly facilitated.

Compositional testing [11], [15] is viewed as one of the most promising directions to bridge the gap between the increasing complexity of systems and actual testing method limits due to the reasons discussed above. Similarly to compositinality result in [15], which establish under certain hypothesis that the conformance testing relation $ioco$ is compositional with respect to parallel composition and hiding, we have established a compositinality result in [2]. This result expresses that for the conformance relation $ioco$[1] and n implementations and specifications iut_i and spec_i, $1 \leq i \leq n$, each one modeled by a component as defined in [2], if for each i, $1 \leq i \leq n$, iut_i $ioco$ spec_i, then for any integration operator of arity n (see Definition 1.7), $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ $ioco$ $op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$. The compositinality result obtained in [2] is thus an extension of *Tretmans*'s result [15] since it is established independently of a given integration operator.

This result justifies the approach that consists in testing separately the components of a system in order to build the correctness of the global system. However, such a result does not help to choose test purposes that are meaningful. Indeed each $\mathsf{iut}_{i_{(i\leq n)}}$ is tested with respect to its specification $\mathsf{spec}_{i_{(i\leq n)}}$, but since testing means selecting a finite number of executions (test cases) to evaluate the conformance, the question is then how to build a meaningful set of executions? Following approaches in [15] and [2], we propose to extract test cases from specifications. However, spec_i, standing alone, does not contain enough information to know how iut_i will be

[1] Actually, a slight extension of this relation to our components called *cioco* in [2] (see Definition 2.1 in this paper).

used in the context of the whole system. This usage is in the end the only aspect that matters at test selection phases since all behaviors reflecting a non-conformance between iut_i and spec_i which are never activated in the context of the whole system $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$, will by definition never cause a fault at the system level. For example, if a system uses a calculator component to invoke only addition, then the component may well be "faulty" for multiplication; this will not cause a fault at the system level. Even more, wasting time to test such behaviors reduces the time and resources to test component behaviors that will be activated in the system frame. This may have dramatically harmful consequences. For example, the disaster of *Ariane* 5 in 1996 is caused by the absence of testing in context of a software component which was only tested for *Ariane* 4. In this paper, we give a new compositionality result that takes into account the behavior of the global system in which components are plugged in. This last result is inspired from the approach proposed in [11], initially developed in the setting of $IOSTS$ (symbolic automaton). In [11], only projection is defined, but no compositinality result is given.

Based on this result, we will then propose a technique that strengthens testing of each component involved in a global system, by choosing suitable test purposes for them. This will be done by defining a projection mechanism that, from global behaviors of a system, will help generating test purposes capturing the behaviors of the subsystems, that typically occur in the context of the whole system.

The paper is structured as follows: Section I recalls our framework for modeling components and systems. Section II introduces the conformance testing theory and discusses its main limitation for the validation of complex software systems. Section III presents the compositionality result and how to test components in the context of the global system.

I. Components and systems

A. Components

In [2], a component is defined as a generalized Mealy automaton in which the dependence between outputs and both current state and inputs is relaxed from a strict determinism, to encompass more complex behaviors such as partiality, non determinism, etc. Components are defined using terminology and notations of coalgebras [17] and monads [3]. Hence, a component in [2] is a coalgebra (S, α) over a signature $T(O \times \_)^I : \mathbf{Set} \longrightarrow \mathbf{Set}$ where T is a monad. The monads have been introduced because they allow us to generically consider many computation situations such as non-determinism, partiality, etc. (see [2], [4] for more explanations). Here, to make the paper easier to read, we restrict ourself to the particular powerset monad (i.e $T = \mathcal{P}$). The generalization to any monad T does not raise any difficulty.

Definition 1.1 (Component): Let I and O be two sets denoting, respectively, the input and output domains. A **component** $\mathcal{C}$ over (I, O) consists of a set of states S and a transition function $\alpha : S \times I \longrightarrow \mathcal{P}(O \times S)$ with a distinguished element $init \in S$ denoting the initial state of $\mathcal{C}$.

Example 1.1: We consider a simple system $\mathcal{S}$ that computes grade averages presented in Fig. 1. This system $\mathcal{S}$ is built from two basic components: a "graphical interface" that helps the user to make various operations on grades and a "calculator" that receives operation commands from the user, performs the requested operation, and reports back to the user.

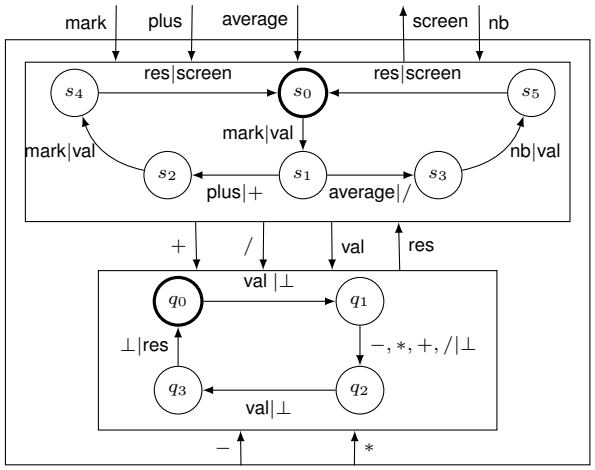

Fig. 1: Grade averages system as a composition of the graphical interface and the calculator

In our framework, the graphical interface is modeled as the component $\mathcal{G} = (\{s_0, s_1, s_2, s_3, s_4, s_5\}, s_0, \alpha_1)$ over the signature

$$\Sigma_1 = (\{\text{mark}, \text{plus}, \text{average}, \text{nb}, \text{res}\}, \{+, /, \text{screen}, \text{val}\})$$

and the calculator as $\mathcal{C} = (\{q_0, q_1, q_2, q_3\}, q_0, \alpha_2)$ over

$$\Sigma_2 = (\{+, *, -, /\}, \{\text{res}, \perp\})$$

α_1 (resp. α_2) is depicted in the box at the top side (resp. bottom side) of Fig. 1.

Definition 1.2 (Component finite traces): The **finite trace** from a state s of a component $\mathcal{C}$, noted $\mathbf{Trace}_{\mathcal{C}}(\mathbf{s})$, is the whole set of the finite input-output sequences $\langle i_0|o_0, \ldots, i_n|o_n\rangle$ s.t $\exists(s_0, \ldots, s_{n+1}) \in S^*$ where

$$\forall j, 0 \leq j \leq n, (o_j, s_{j+1}) \in \alpha(s_j)(i_j) \text{ with } s_0 = s$$

Hence, **the set of traces of** $\mathcal{C}$, noted $Trace(\mathcal{C})$, is the set $Trace_{\mathcal{C}}(init)$.

In the following, we note $\alpha(s)(i)_{|_1}$ (*resp.* $\alpha(s)(i)_{|_2}$) the set composed of all first arguments (*resp.* second arguments) of couples in $\alpha(s)(i)$.

B. Systems

Larger systems are built by composition from two basic operators: *Cartesian product* and *feedback*.

Cartesian product: The cartesian product is a composition where both components are executed simultaneously when triggered by a pair of input values.

Definition 1.3 (Cartesian product $\otimes$): Considering two components $\mathcal{C}_1 = (S_1, s_1^0, \alpha_1)$ and $\mathcal{C}_2 = (S_2, s_2^0, \alpha_2)$ over (I_1, O_1) and (I_2, O_2) resp. The **Cartesian product** $\mathcal{C}_1 \otimes \mathcal{C}_2$

of $\mathcal{C}_1$ and $\mathcal{C}_2$ is the component $(S_1 \times S_2, (s_1^0, s_2^0), \alpha)$ over $(I_1 \times I_2) \times (O_1 \times O_2)$ where α is the mapping defined for every $(i_1, i_2) \in I_1 \times I_2$ and every $(s_1, s_2) \in S$ by:

$$\alpha((s_1,s_2))((i_1,i_2)) = \big\{((o_1,o_2),(s_1',s_2'))| \\ (o_k, s_k') \in \alpha(s_k)(i_k) \text{ for } k = 1,2\big\}$$

Feedback: The concept of feedback composition is intrinsic in dynamic system modeling in control theory [12]. Here, we fit it to discrete systems. A component with *feedback* has directed cycles, where an output from a component is fed back to affect an input of the same component. That means the output of a component in any feedback composition depends on an input value that in turn depends on its own output value.

First, we introduce feedback interfaces for defining correspondences between outputs and inputs of components and keeping only inputs and outputs that are not involved in the feedback.

Definition 1.4 (Feedback interface): A **feedback interface** over an interface signature (I, O) is a triplet $\mathcal{I} = (f, \pi_i, \pi_o)$ where $f : I \times O \longrightarrow I$ is a mapping, and $\pi_i : I \longrightarrow I'$ and $\pi_o : O \longrightarrow O'$ are surjective mappings such that $\forall (i,o) \in I \times O, f(f(i,o),o) = f(i,o)$ *and* $\pi_i(i) = \pi_i(f((i,o)))$.

The mapping f specifies how components are linked and which parts of their interfaces are involved in the composition process. It finds the new value of the input that it is both a valid input and a valid output of the component, given its current state. Both mappings π_i and π_o can be thought as extensions of the hiding connective found in process calculi [13].

The feedback operator we consider here is *synchronous*. That means at some reaction r, the output of component $\mathcal{C}$ in r must be available to its inputs in the same reaction r. The synchronous feedback requires then the existence of an instantaneous fix-point that gives rise to the notion of *well-formed feedback interface*.

Definition 1.5 (Well-formed feedback interface): Let $\mathcal{C}$ be a component over $\Sigma = (I, O)$ and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over Σ. We say that $\mathcal{I}$ is **well-formed w.r.t** $\mathcal{C}$ if, and only if $\forall s, s \in S$ and $\forall (x_1, \ldots, x_n) \in I^*$, $\exists (y_1, \ldots, y_n) \in O^*$ s.t$\forall j, 1 \leq j < n,\ y_j \in \alpha(s)(f(x_j, y_j))_{|_1}$.

Definition 1.6 (Synchronous feedback $\circlearrowleft$): Let $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over $\Sigma = (I, O)$. Let $\mathcal{C} = (S, init, \alpha)$ be a component over Σ such that $\mathcal{I}$ is well-formed w.r.t $\mathcal{C}$. The **synchronous feedback over** $\mathcal{I}$, $\circlearrowleft_{\mathcal{I}}(\mathcal{C})$, is the component $\mathcal{C}' = (S, init, \alpha')$ over $\Sigma' = (I', O')$ where α' the mapping defined for every $s \in S$ and every $i' \in I'$ by: $\alpha'(s)(i') = \big\{(o', s') | \exists (i,o) \in (I \times O), (o, s') \in \alpha(s)(f(i,o)), \pi_i(i) = i' \ and \ \pi_o(o) = o'\big\}$

Complex operators and systems:
As previously explained, from Cartesian product and feedback operators, we can build more complex ones by composition.

Definition 1.7 (Complex operator): The **set of complex operators**, is inductively defined as follows:

- _ is a complex operator of arity 1;
- if op_1 and op_2 are complex operators of arity n_1 and n_2 respectively, then $op_1 \otimes op_2$ is a complex operator of arity $n_1 + n_2$;
- if op is complex operator of arity n and $\mathcal{I}$ is a feedback interface, then $\circlearrowleft_{\mathcal{I}}(op)$ is a complex operator of arity n.

Complex operators will not be necessarily defined when applied to a sequence of components. Indeed, for a complex operator of the form $\circlearrowleft_{\mathcal{I}}(op)$, according to the component $\mathcal{C}$ resulting from the evaluation of op, the interface $\mathcal{I}$ has to be defined over the signature of $\mathcal{C}$ and the feedback over $\mathcal{C}$ has to be well-formed.

Definition 1.8 (Systems): Let $\mathbb{C}$ be a set of components. The **set of systems over** $\mathbb{C}$ is inductively defined as follows:

- for any $\mathcal{C} \in \mathbb{C}$, a component over a signature Σ, $\_(\mathcal{C}) = \mathcal{C}$ is a system over the signature Σ and _ is **defined for** $\mathcal{C}$;
- if $op_1 \otimes op_2$ is a complex operator of arity $n = n_1 + n_2$ then for every sequence $(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1}, \ldots, \mathcal{C}_n)$ of components with each $\mathcal{C}_i$ over (I_i, O_i), if both op_1 and op_2 are defined for $\mathcal{C}_1, \ldots, \mathcal{C}_{n_1}$ and $\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n$ resp., then $op_1 \otimes op_2(\mathcal{C}_1, \ldots, \mathcal{C}_n) = op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1}) \otimes op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$ is a system over $(\prod_{i=1}^n I_i, \prod_{i=1}^n O_i)$ and $op_1 \otimes op_2$ is **defined for** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, else $op_1 \otimes op_2$ is not.
- if $\circlearrowleft_{\mathcal{I}}(op)$ is a complex operator of arity n, then for every sequence $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ of components, if op is defined for $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, $\mathcal{I}$ is a feedback interface over Σ and $\mathcal{I}$ is well-formed w.r.t $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, then $\circlearrowleft_{\mathcal{I}}(\mathcal{S})$ is a system over Σ' and[2] $\circlearrowleft_{\mathcal{I}}(op)$ is **defined for** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, else $\circlearrowleft_{\mathcal{I}}(op)$ is not.

We introduce the definition of a subsystem involved in a system. This intuitively allows us to characterize the set of all basic subsystems from which the global system can be built.

Definition 1.9 (Sub-systems): Let $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ be a system over Σ. The **set of sub-systems of** $\mathcal{S}$, noted $\mathbb{S}ub(\mathcal{S})$, is inductively defined on the structure of op as follows:

- if $op = \_$, then $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\}$;
- if $op = op_1 \otimes op_2$ with op_1 and op_2 of arity n_1 and n_2 respectively (i.e. $n = n_1 + n_2$), then $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\} \cup \mathbb{S}ub(op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})) \cup \mathbb{S}ub(op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n))$;
- if $op = \circlearrowleft_{\mathcal{I}}(op')$, then
$$\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\} \cup \mathbb{S}ub(op'(\mathcal{C}_1, \ldots, \mathcal{C}_n))$$

Example 1.2: The system $\mathcal{S}$ to compute grade averages is obtained as a composition of $\mathcal{G}$ and $\mathcal{C}$ using our basic integration operators. To define $\mathcal{S}$, we first apply the Cartesian product $\otimes((\mathcal{G}, \mathcal{C}))$ to $\mathcal{G}$ and $\mathcal{C}$ over the signature $\Sigma_\otimes = (I_\otimes, O_\otimes)$ with: $I_\otimes = (\{\text{mark}, \text{plus}, \text{average}, \text{nb}\} \times \{\text{val}, +, /\})$ and $O_\otimes = (\{\text{val}, \text{screen}, \perp\} \times \{\perp, \text{res}\})$. We can then see that:

- both outputs $+$ and $/$ of $\mathcal{G}$ are returned as inputs of $\mathcal{C}$;
- the output "res" of $\mathcal{C}$ is returned as input of $\mathcal{G}$.

Then, we apply the synchronous feedback $\circlearrowleft_{\mathcal{I}}$ to $\otimes((\mathcal{G}, \mathcal{C}))$ over the interface signature $\mathcal{I} = (f, \pi_i, \pi_o)$ as follows:

[2] Σ' is the signature of the synchronous feedback.

$$f : I_{\otimes} \times O_{\otimes} \longrightarrow I_{\otimes}$$
$$((i,i'),(o,o')) \mapsto \begin{cases} (i,o) \text{ if } i' = o \\ (i,i') \text{ otherwise} \end{cases}$$
$$\pi_i : I_{\otimes} \longrightarrow I_{\mathcal{G}} \cup I_{\mathcal{C}} \qquad \pi_o : O_{\otimes} \longrightarrow O_{\mathcal{G}} \cup O_{\mathcal{C}}$$
$$(i,i') \mapsto \begin{cases} i \text{ if } i' \in O_{\mathcal{C}} \\ i' \text{ otherwise} \end{cases} \quad (o,o') \mapsto \begin{cases} o' \text{ if } o \in I_{\mathcal{G}} \\ o \text{ otherwise} \end{cases}$$

This leads to a new component $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G},\mathcal{C}))$ (see Fig. 2) where all outputs of $\mathcal{G}$ (i.e $+, /$ and val) that are fed back to $\mathcal{C}$ and the output "res" of $\mathcal{G}$ that is fed back to $\mathcal{C}$ are hidden.

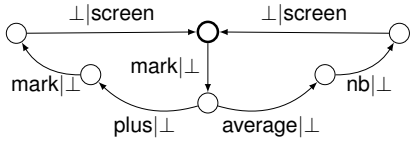

Fig. 2: Component $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G},\mathcal{C}))$

II. Conformance testing

Conformance testing theory is usually based on the comparison between the behavior of a specification and an implementation using a conformance relation. The goal of this relation is to specify what the conformance of an implementation is with respect to its specification. It has been shown that the input-output conformance relation *cioco* is the most suitable for testing our components [2]. This relation distinguishes input and outputs actions, and requires that the implementation behaves according to a specification, but also allows behaviors on which the specification puts no constraint.

The specification spec of a component is the formal description of its behavior given by a component over a signature (I,O). On the contrary, its implementation iut is an executable component, which is considered as a black box [18]. We interact with the implementation through its interface, by providing inputs to stimulate it and observing its behavior through its outputs. Hence, to be able to treat the implementation iut, we make the following two assumptions about it:

- The implementation iut can be modeled as a component $(S, init, \alpha)$ over the signature (I',O') with $I \subseteq I'$ to allow the implementation to accept all the inputs of the[3] specification and $O' \subseteq O$ to allow the specification to accept all the responses of the implementation.
- iut is **input-enabled**, i.e. at any state, it must produce answers for all inputs provided by the environment: $\forall (s,i) \in S \times I, \exists (o,s') \in O \times S$ s.t $(o,s') \in \alpha(s)(i)$

The conformance relation that we will call here *cioco*[4] is a slight adaptation of the standard relation *ioco* [9].

Definition 2.1: (cioco) Let spec, iut be two components over (I,O) and (I',O') resp. such that $I \subseteq I', O' \subseteq O$ and iut is input-enabled. iut **is in conformance with** spec, noted iut *cioco* spec, if and only if: $\forall tr \in Trace(\mathsf{spec}), \forall i \in I$

$$Out(\mathsf{iut} \text{ after } (tr,i)) \subseteq Out(\mathsf{spec} \text{ after } (tr,i))$$

where for any component $\mathcal{C}$, any finite trace tr, and any input i of $\mathcal{C}$, $Out(\mathcal{C} \text{ after } (tr,i))$ is the set $\{o \mid tr.\langle i|o\rangle \in Trace(\mathcal{C})\}$.

Similarly to [15], we studied in [2] compositionality properties for *cioco* over integration operators defined in Section I-B. We then proved the following theorem:

Theorem 2.1 (Compositionality [2]): Let op be a complex operator of arity n. Let $\mathsf{iut}_1, \ldots, \mathsf{iut}_n, \mathsf{spec}_1, \ldots, \mathsf{spec}_n$ be input-enabled components. If $\forall i, 1 \leq i \leq n$, iut_i *cioco* spec_i, then one has $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ *cioco* $op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$.

This means that if every single component of a system conforms to its specification, the whole system built over our integration operators conforms to its specification, if the specification is input-enabled. Such a testing compositionality result provides a way to test the integrated system only by testing its subsystems. Thus, once this property is verified, the correctness of the integrated system is obtained from the correctness of the individual components. To test the integrated system, it is not necessary to consider it as a whole, but it is enough to consider its sub-systems and test them separately. Indeed, the contraposition of this property is the following:

$$\neg\Big(op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n) \textit{ cioco } op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)\Big) \Longrightarrow \exists i, 1 \leq i \leq n, \neg(\mathsf{iut}_i \textit{ cioco } \mathsf{spec}_i)$$

Thus, by looking at this new property, we can easily see that non-correctness of the integrated system under test $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ implies that at least one of its components $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ is incorrect.

In the sequel, we will show how to improve significantly the result obtained in Theorem 2.1 by taking into account the global system in which components are plugged in. This will be achieved by using projection mechanisms.

III. Projection and test purposes

A. Projection and compositionality

Projection techniques [11] are defined by pruning from any global behavior p, all that does not concern the subsystem that we want to test. This allows us to generate more relevant unit test cases to test individual components. As an illustration, let us consider again the system that computes grade averages (see Example 1.2). According to the result obtained in Theorem 2.1, to test the grade average system, it suffices to test separately the calculator $\mathcal{C}$ and the controller $\mathcal{G}$. Now, testing the calculator $\mathcal{C}$ separately may lead to the consideration of test cases involving arithmetic operations which are irrelevant to computing student grade averages such as subtraction or multiplication. This may cause test cases of interest to the system to be missed, *i.e.* test cases only bringing into play addition and division for grades ranging from 0 to 20. In the approach we propose in the following, we intend to generate a test purpose that guides the test derivation process of $\mathcal{C}$ by only testing operations needed to compute grade averages. We do this by making a projection of this behavior on calculator component $\mathcal{C}$.

[3] I and O are the input and output sets of the specification respectively.
[4] c for component

Definition 3.1 (Projection): Let $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ be a system over (I, O). Let $sub \in \mathbb{S}ub(\mathcal{S})$ be a sub-system of $\mathcal{S}$ over (I', O'). Let $tr = \langle i_1|o_1, \ldots, i_m|o_m\rangle \in Trace(\mathcal{S})$. The **projection of** tr **on** sub, denoted by $tr_{\downarrow_{sub}}$, is the subset of $Trace(sub)$ inductively defined as follows:

- if $op = \_$, then $tr_{\downarrow_{sub}} = \{tr\}$;
- if $op = op_1 \otimes op_2$ with op_1 and op_2 of arity n_1 and n_2 respectively (i.e. $n = n_1 + n_2$), then[5]:

$$tr_{\downarrow_{sub}} = \begin{cases} \text{is the projection of } \langle i_{1_{|_1}}|o_{1_{|_1}}, \ldots, i_{m_{|_1}}|o_{m_{|_1}}\rangle \\ \text{on sub if } sub \in \mathbb{S}ub(op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})) \\ \text{is the projection of } \langle i_{1_{|_2}}|o_{1_{|_2}}, \ldots, i_{m_{|_2}}|o_{m_{|_2}}\rangle \\ \text{on sub otherwise.} \end{cases}$$

- if $op = \circlearrowleft_{\mathcal{I}}(op')$ with $\mathcal{I} = (f, \pi_i, \pi_o)$, then

$$tr_{\downarrow_{sub}} = \bigcup_{tr' \in tr_{\downarrow_{\mathcal{S}'}}} tr'_{\downarrow_{sub}} \text{ where}$$

$\mathcal{S}' = op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ and

$tr_{\downarrow_{\mathcal{S}'}} = \{\langle i'_1|o'_1, \ldots, i'_m|o'_m\rangle \mid \forall j, 1 \leq j \leq m, \exists s_j \in S',$
$o'_j \in \alpha_{\mathcal{S}'}(s_j)(f(i'_j, o'_j))_{|_1}, i_j = \pi_i(i'_j)$
and $o_j = \pi_o(o'_j)\}$

We then introduce the projection of a system on one of its sub-systems.

Definition 3.2 (Component in context): Let $\mathcal{S}$ be a system over (I, O) and $sub \in \mathbb{S}ub(\mathcal{S})$ be a subsystem of $\mathcal{S}$ over (I', O'). The **component obtained by projecting** $\mathcal{S}$ **on** sub, noted $\mathcal{S}_{\downarrow_{sub}}$ is the triplet (S, s^0, α) defined by:

- $s^0 = \langle\rangle$
- S is the whole set of finite traces defined as follows:
  - $s^0 = \{\langle\rangle\}$
  - $\forall j, 1 \leq j \leq n, s^j = \{tr'.\langle i|o\rangle \mid \exists tr' \in s^{j-1}, \exists i \in I', \exists o \in O', \exists tr \in Trace(\mathcal{S})\ s.t\ tr'.\langle i|o\rangle \in tr_{\downarrow_{sub}}$

  Hence, $S = \bigcup_{0 \leq j \leq \omega} s^i$
- $\alpha : S \times I' \longrightarrow \mathcal{P}(O' \times S)$ is the mapping which associates with every $\langle i_0|o_0, \ldots, i_m|o_m\rangle \in S$ and every input $i \in I'$ the set:

  $\Pi = \{(o, \langle i_0|o_0, \ldots, i_m|o_m, i|o\rangle) \mid \exists o \in O', \exists tr \in Trace(\mathcal{S})$ with $\langle i_0|o_0, \ldots, i_m|o_m, i|o\rangle \in tr_{\downarrow_{sub}}\}$

Example 3.1: Fig. 3 shows the projection $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))_{\downarrow_{\mathcal{C}}}$ of $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))$ on the calculator $\mathcal{C}$. By applying Def. 3.2, we only retain the behaviors of $\mathcal{C}$ that are involved in the final behavior of $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))$. Only the addition and the division operations are specified in $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))_{\downarrow_{\mathcal{C}}}$, the specifications of both subtraction and multiplication operations are omitted due to their absence in the global system $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))$.
Such projected traces will be the cornerstone to improve the compositionality result presented in Theorem 2.1 and to define test purposes dedicated to test components separately while taking into account the behavior of the global system.

Theorem 3.1 (Compositionality with projection): Let op be a complex operator of arity n. Let $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ be input-enabled implementations and $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$ their specifications respectively. Then, one has $\forall i, 1 \leq i \leq n$

[5] $a_{|_i}$ is the projection of the n-tuple a on i^{th} argument.

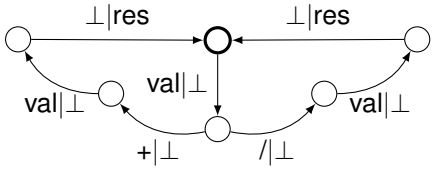

Fig. 3: The projection of $\circlearrowleft_{\mathcal{I}}((\mathcal{G}, \mathcal{C}))$ on the calculator $\mathcal{C}$

$\forall i, 1 \leq i \leq n, \mathsf{iut}_i\ cioco\ op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)_{\downarrow_{\mathsf{spec}_i}}$
$\Longrightarrow op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)\ cioco\ op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$

The proof of the theorem is available in an extended version of this paper [19].

Theorem 3.1 then provides a way to test the integrated system only by testing the projection of that system on its sub-systems. As a consequence, to test the integrated system, it is not necessary to consider it as a whole, but it is enough to consider the projection of that system on its sub-systems (which may be done at different development steps and eventually developed by different teams) and test them separately.

Comparing this result with our previous result presented in [2] or *Tretmans*'s result [15], the new result does not require that the specifications are input-enabled. This last property is often hard to get in practice due to the fact that system input domains are usually too large.

B. Test purpose

A specification model usually contains a growth of exponential states which makes the testing process difficult or even impossible to implement. To cope with this problem, test purposes can be used. A test purpose is a description of the part of the specification that we want to test and for which test cases are later generated. In [10], they are described independently of the model of the specification. In [16], they are deduced from the specification by construction. In order to guide the test derivation process in our approach, we have preferred, as in [16], to describe test purposes by selecting the part of the specification that we want to explore. We therefore consider a test purpose as a tagged finite computation tree (FCT) of the specification. The leaves of the FCT which correspond to paths that we want to test are tagged accept. All internal nodes on such paths are tagged skip, and all other nodes are tagged $\odot$.

Definition 3.3 (Finite computation tree of a component): Let (S, s_0, α) be a component over (I, O). The **finite computation tree** of depth n of $\mathcal{C}$, noted $FCT(\mathcal{C}, n)$, is the triplet $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ defined by:

- S_{FCT} is the whole set of $\mathcal{C}-$paths. A $\mathcal{C}-$path is defined by a couple $\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle$ such that:
$$\forall j, 1 \leq j \leq n, s_j \in \alpha(s_{j-1})(i_{j-1})_{|_2}$$
- s^0_{FCT} is the initial $\mathcal{C}-$path $\langle s_0, ()\rangle$
- α_{FCT} is the mapping which associates with every $\mathcal{C}-$path $\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle$ and every input $i \in I$ the set:
$\Gamma = \{(o, \langle(s_0, \ldots, s_n, s'), (i_0, \ldots, i_{n-1}, i)\rangle)$
$\mid (o, s') \in \alpha(s_n)(i)\}$

In this definition, S_{FCT} is the set of the nodes of the tree and s^0_{FCT} its root. Each node is represented by the unique $\mathcal{C}$-path $\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ which leads to it from the root. α_{FCT} gives, for each node p and for each input i, the set of nodes Γ that can be reached from p when the input i is submitted to $\mathcal{C}$.

In the following, we intend to extend the notion of test purpose proposed in [2] to test purpose in context. This allows us to test, from a global behavior of a system, the behavior of its involved sub-systems and then guide the component testing intelligently by taking into account the way components are used in systems. Thus, taking a behavior p of a system $\mathcal{S}$, we intend to define test purposes that are able to test the behavior p_i of each sub-system $\mathcal{S}_i \in \mathbb{S}ub(\mathcal{S})$. We identify therefore for each sub-system all its finite paths that are involved in constructing the whole behavior of $\mathcal{S}$.

Definition 3.4 (Test purpose in context): Let $\mathcal{S}$ be a system over (I, O). Let $sub \in \mathbb{S}ub(\mathcal{S})$ be a sub-system of $\mathcal{S}$ and $sub' = \mathcal{S}_{\downarrow_{sub}}$ the projection of $\mathcal{S}$ on sub. Let $FCT(sub, n) = (S, s_0, \alpha)$ be a finite computation tree of sub. A **test purpose in context** TP for sub is a mapping

$$TP : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\} \text{ such that:}$$

- for every node $p = \langle i_0 | o_0, \ldots, i_m | o_m \rangle \in Trace(sub')$, $TP(p) = \mathsf{accept}$;
- if $TP(\langle i_0 | o_0, \ldots, i_m | o_m) \rangle = \mathsf{accept}$, then: $\forall j, 0 \leq j \leq m, TP(\langle i_0 | o_0, \ldots, i_{j-1} | o_{j-1}) \rangle) = \mathsf{skip}$
- $TP(\langle \rangle) = \mathsf{skip}$
- if $TP(\langle i_0 | o_0, \ldots, i_k | o_k) \rangle) = \odot$, then: $\forall k < k' \leq n, \forall (i'_l)_{k \leq l < n} \in I',$ and $\forall (o'_l)_{k \leq l < n} \in O'$ $TP(i_0 | o_0, \ldots, i_k | o_k, i'_{k+1} | o'_{k+1}, \ldots, i'_{k'} | o'_{k'}) \rangle) = \odot$

In order to build a test purpose for a subsystem sub, we identify all finite paths of its finite computation tree FCT whose traces embody traces in $Trace(sub')$ and we tag them with accept. We then tag every node which represents a prefix of an accepted behavior with skip. The other nodes, which lead to behaviors that we do not want to test, are tagged with $\odot$.

Example 3.2: We intend to build a test purpose dedicated to test the behavior of the calculator component $\mathcal{C}$ in the context of the system computing grade averages. To do so, we first build the finite computation tree $FCT(\mathcal{C}, 4)$ that we present in Fig. 4. Second, each state of $FCT(\mathcal{C}, 4)$ that is reachable after each trace of $\circlearrowleft_{\mathcal{I}}(\otimes(\mathcal{G}, \mathcal{C}))_{\downarrow_{\mathcal{C}}}$ (see Fig. 2) is tagged with accept. Only p_{10} and p_{12} are tagged with accept. All nodes leading from the root p_0 to p_{10} or p_{12} are tagged with skip (i.e p_2, p_4, p_6 and p_8), and all other nodes are tagged with $\odot$.

IV. Conclusion

This paper extends our previous work [2] which defines a generic testing conformance theory. We have proposed an approach to test components that are typically involved in the whole system by defining test purposes from the global behaviour of the whole system. Such test purposes are given in an accurate way by defining a projection mechanism taking

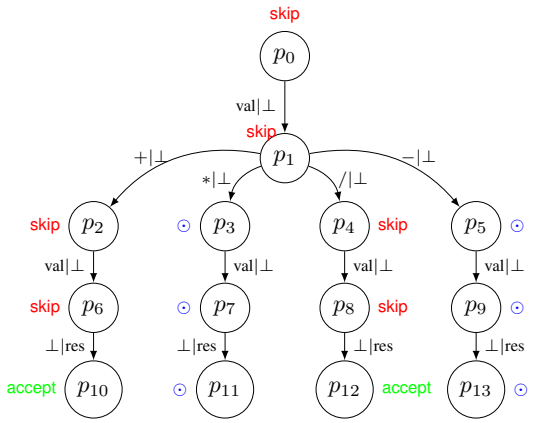

Fig. 4: Test purpose of the calculator component

a global behaviour p of the whole system and keeping only the part of p being activated in the sub-system that we want to test. Thus, our method for generating test purposes from the global system specification helps to generate relevant unit test cases to test individual components.

September 20, 2012

References

[1] D'Souza, D.F. and Wills, A.C., *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach* Addison-Wesley Prof., 1998.

[2] M. Aiguier, F. Boulanger and B. Kanso, *A formal abstract framework for modeling and testing complex software systems*, Theoretical Computer Science (TCS), 455, 66-97, Elsevier, 2012.

[3] S. MacLane, *Categories for the Working Mathematician*, Springer Verlag, Graduate Texts in Mathematics, New York, Heidelberg, Berlin, 1971.

[4] E. Moggi, *Notions of computation and monads*, Information and Computation journal, 93, 55-92, 1991.

[5] L.S Barbosa, *Towards a Calculus of State-based Software Components*, Journal of Universal Computer Science, 9(8):891-909, August 2003.

[6] Meng, S. and Barbosa, L.S., *Components as coalgebras: the refinement dimension*, Theoretical Computer Science (TCS), 351(2):276-294, Elsevier Science Publishers Ltd, Essex,UK, 2006.

[7] G. H. Mealy, *A method for synthesizing sequentiel circuits*, Bell Systems Techn. Jour. journal, 0167-6423, 1955.

[8] R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, New York, Inc, secaucus, NG, USA, 1982.

[9] J. Tretmans, *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation*, Computer networkss and ISDN systems, 29(1):49-79, 1996.

[10] C. Jard, T. Jéron, *TGV: theory, principles and algorithms*, International Journal on Software Tools for Technology Transfer,7(4):297–315, 2005.

[11] A. Faivre, C. Gaston and P. Le Gall, *Symbolic Model Based Testing for Component Oriented Systems*, , TestCom/FATES, 90–106, 2007.

[12] Edward A. Lee and Pravin Varaiya, *Structure and interpretation of signals and systems*, Addison-Wesley,I-XXI, 1-647, 2003.

[13] C. A. R. Hoare, *Communicating Sequential Processes*, journal of Communications of the ACM, 21: 666-677 1985.

[14] G. Gossler and J. Sifakis. *Composition for component-based modeling*, Sci. Comput. Program., 55(1-3):161183, 2005

[15] H.M. van der Bijl, A. Rensink and J. Tretmans, *Compositional Testing with ioco*, FATES,LNCS, 2931:86–100, Berlin, 2004.

[16] C. Gaston, P. Le Gall, N. Rapin and A. Touil, *Symbolic Execution Techniques for Test Purpose Definition*, TestCom, 1-18, 2006.

[17] J. Rutten, *Universal coalgebra: a theory of systems*, Theoretical Computer Science (TCS), 249(1), pages: 3-80, 2000.

[18] J. Tretmans, *A Formal Approach to Conformance Testing*, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, 257–276, Amsterdam, The Netherlands, 1994.

[19] B. Kanso. *Conformance testing of component-based systems*, Internal report 2012-09-13-DI-FBO, Suplec, 2012. http://wwwdi.supelec.fr/internalreports/InternalReport-2012-09-13.pdf.