A Formal Methodology to Test Complex Heterogeneous Systems*

Ismael Rodríguez and Manuel Núñez

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
{isrodrig,mn}@sip.ucm.es

Abstract. Complex computational systems may integrate heterogeneous components that may be defined in different ways. In order to test the conformance of an implementation we can use a different testing methodology for each of the (different groups of) components. Still, this approach might overlook details regarding the relation among the parts of the system under test. We present an integrated testing methodology that takes into account the hierarchical dependence of all the parts of a system. Its main peculiarity is that parts of the *implementation under test* (IUT) that have already been tested may partially *define* the behavior of the tests used to check the correctness of other IUT parts.

1 Introduction

The complexity and heterogeneity of systems has reached a level where old solutions to assess reliability turn out to be obsolete. To overcome this problem, *formal testing techniques* [3,5,1,6,7] provide systematic procedures to create and apply tests to implementations. Usually, these techniques require constructing a formal specification to precisely define the expected capabilities of the system. Then, these capabilities are contrasted with those of the implementation. The task of completely specifying the behavior of current computational systems is very difficult due to their complexity. In particular, most real systems are very heterogeneous and include a big amount of components with different natures. Thus, instead of using a unique specification framework to formally define the behavior of systems, it is more adequate to define distinguished parts of the system by using different formalisms. In addition to be composed by several parts, systems often present a *hierarchical* structure. Thus, its definition can be decomposed into different levels of abstraction, consisting each of these levels of several units. In this case, the behavior of each unit will be formally specified by using a (possibly) different formalism. The formal specification of complex heterogeneous systems, by using different formalisms, represents an important challenge for traditional testing methodologies.

The purpose of this paper is to introduce a formal methodology to test this kind of systems. First, we consider that specifications are not described by using

* Research partially supported by the Spanish MEC project WEST/FAST TIN2006-15578-C02-01 and the Marie Curie RTN *TAROT* (MRTN-CT-2003-505121).

a unique language but that several languages can be used to specify different (sets of) functionalities, that is, different units. Systems have a hierarchical structure (different levels). Each level consists of several units. The first level contains those units that do not depend on other units. For any $i > 1$, the i-th level contains those units that depend only on units belonging to lower levels. Even though the specification of multi-level systems is a stand-alone goal, in this paper we will go one step further. Our objective is to test the conformance of IUTs with respect to these specifications. Obviously, even if a system is specified in different stages, so that the process can be somehow viewed as working with several specifications, we will have a unique integrated implementation, not one implementation for each of the units/levels. In fact, this implementation will be often given in terms of a *black box*, where the internal structure is not observable. Thus, we need tests that can stimulate the IUT by means of basic low level operations that can be understood by it. An obvious mechanism to test these kind of systems can be used if each unit of the IUT can be *disconnected* from those units it depends on. If this is possible then we could test different parts of the IUT in an *isolated* way. Unfortunately, it is not always feasible to disconnect parts of the IUT from other parts they depend on directly. In particular, this process could require some knowledge of the IUT that is not available if the IUT is a black box. Besides, we lose the capability to check the correctness of the *integration* and *assembling* of the parts of the IUT.

We present an integrated testing methodology to test multi-level systems. Since specifications are defined by several levels (and possibly with different formalisms) tests will be so. Different tests to check the correctness of the system at each level will be provided and all levels will be tested bottom-up. The main peculiarity of the methodology is that, once a level has been tested and no failure is found, that IUT level will provide part of the behavior of the tests that will be used to test *upper* levels. Let us consider a system defined in terms of three hierarchical levels. A test devoted to check the intermediate level interacts with the IUT by performing operations that may be defined in terms of other lower level operations. So, in order to stimulate the IUT, this test needs to perform some lower level operations as well. Once all lower level operations have been validated in the IUT by a suitable test suite (consisting of some lowest level tests), intermediate level tests will be able to perform each needed lower level operation by *invoking* the corresponding implementation of this operation in the IUT. That is, part of the behavior of tests will be directly given by the IUT (once the corresponding IUT part has been tested). Besides, the methodology will be refined to deal also with *circular dependencies* between units.

Let us note that our approach contains, as particular cases, standard approaches for the specification and testing of component based systems. Actually, each testing methodology designed to work with a formalism may provide a suitable mechanism to deal with the corresponding level of a multi-level specification. Each of these methodologies will be properly integrated in our methodology. Besides, protocols for testing multi-level systems such as ISO 9646 [2] are also easily included in our methodology. In fact, in [2] systems are *linear* in the sense that

there are n levels where each level contains a unique unit. This is a much simpler conception of systems than the one we present in this paper. Besides, tests do not have the capability of using the IUT as part of their own definition and environments with circular dependencies between units are not considered.

Our testing approach is presented informally and formally in the next section and in Section 3, respectively. In Section 4 we extended it to deal with *circular dependencies* among units. Some conclusions are given in Section 5.

2 Informal Presentation of the Methodology

In this section we informally introduce the main characteristics of our approach. We will review how traditional formal testing works and we will show how it can be adapted/modified/discarded in the case of multi-level testing. In formal testing we usually extract some tests from the considered specification and apply them to the implementation under test (IUT). By comparing the obtained results with the ones expected by the specification we can generate a *diagnosis* about the (in-)correctness of the IUT. These tests usually represent *stimulation plans* of the IUT by providing sequences of events that can be interpreted by the IUT. If the specification language specializes in defining simple communication events (e.g. the messages of a communications protocol) then these events can be straightforwardly reproduced and identified. In this case, it will be easy to apply the tests to the IUT, since the events clearly define how to stimulate the IUT. On the contrary, if the specification language deals with a higher level of abstraction then it will be more complex to decide how the test will apply the foreseen plan of events to the IUT.

Let us consider the following example. We specify an autonomous e-commerce agent by using a language that specializes only in the definition of its highest level of abstraction, that is, the economic objectives of the agent (e.g. [4]). The resulting model will clearly define, for each situation, the set of desirable transactions. Thus, the specification of an agent will indicate its behavior after receiving a given offer, as well as the way in which the agent will propose offers to other agents. Let us suppose that this high-level specification language does not define how an exchange proposal is split into simpler operations. In such a context, one may wonder how a test can indicate to the IUT that a transaction has to be performed. If the specification language deals only with high level behavior then the test should not interact with the implementation by following a fixed communications protocol. This is so because the specification language does not allow to express these details. Thus, if we fix the e-commerce protocol used by the test then we are forcing the IUT to follow that protocol, even though this information is not reflected in the specification. However, it is obvious that the test will be using a specific protocol. Otherwise, it would not be able to propose transactions to the IUT, being these the basic operations to stimulate the IUT. In order to solve the previously stated problem we will impose two conditions.

First, we need a more complete definition of the specification where lower abstraction levels are somehow included. Let us remark that even though a high level language can be used to provide key information about the desired behavior of the agent, the agent must indicate some lower level operations allowing to perform the desired functionalities. Thus, the specification of the agent must be done in several steps and by (potentially) using different specification languages for each level of abstraction. For example, a lower level defining the e-commerce communication protocol must be included in the agent specification.

Second, the definition of tests will be also given by levels/units. The tester has to use the same levels of abstraction as the ones in the specification. In order to test a certain level of abstraction of the IUT the tests will represent activity plans for that level. Nevertheless, each of the interactions between the test and the IUT will be carried out by using auxiliary (possibly lower level) operations. For example, if a test sends a *resource exchange offer* to the IUT then this operation requires to open a channel, codifying the exchange according to some protocol, sending the offer through the channel, etc. Lower level operations have to be tested beforehand with respect to the lower levels of abstraction where they are defined. Besides, lower levels of abstraction are tested by using operations of other lower levels of abstraction, and so on. In general, the definition of a test to check the behavior of the IUT at a level of abstraction needs the definition of all the levels of the test from that level down to the lowest level.

Thus, in order to define tests we will use a bootstrapping approach. The behavior of the units belonging to the lowest level will be tested as usual. That is, tests interact with the IUT by using atomic operations that are understood by both parts and no further definition is required. The difference comes when testing the behavior of a higher unit and, more precisely, when defining tests for these units. If a test is devoted to check a unit belonging to a certain level of the IUT then it needs to use lower level operations to interact with the IUT. However, the test does not have its own definition of how to split operations at level i into operations at level $i-1$. Instead, this definition will be taken from the IUT: The test will invoke IUT operations belonging to lower levels as part of its own definition. Let us note that the purpose is *not* to test these $(i-1)$-level IUT operations, but to *use* them to properly interact with the IUT at level i. That is, part of the test behavior (specifically, that concerning to the use of operations of level $i-1$) will be directly given by the IUT. However, before we use these IUT operations, we need to be confident that they are correctly implemented; otherwise tests would not work as planned. Thus, we require that these IUT operations have already been tested by other (lower level) tests.[1] These lower level tests may need to use IUT operations of other lower levels, and so on. This procedure yields a recursive testing methodology where units belonging to lower levels must be tested before the ones corresponding to higher levels.

[1] Even if these lower IUT operations successfully pass a suitable test suite, their correctness is not guaranteed in general. However, our criterion to assure the correctness of (part of) the tests using them is the *same* as to assure the correctness of IUTs. Hence, if the testing procedure for the IUT is suitable then it will be so for tests.

3 Basic Definitions

In this section we introduce some notation to formally define *multi-level* specifications and tests. Intuitively, a *specification* can be seen as a set of services, that is, functionalities that the system is supposed to provide. Each service is defined by means of an expression in a certain specification language. This expression indicates the operations that take place to perform that service. A given service can depend on other services provided by the specification and/or by other lower level sub-specifications. In particular, a service can be defined by using another service, which in turn is defined in terms of a third service, and so on. In this way, services can define dialogs between the system and the environment. The organization of services in units allows to precisely define how a unit depends on other units. For the sake of notation simplicity, we will assume that some operations over sets can be applied to *tuples* when the order of elements is not relevant. In this case, the operation will transform any tuple $(a_1, \ldots, a_n)$ into a set $\{a_1, \ldots, a_n\}$. For example, we will write expressions such as $e \in (a, b, c) \cup (e, f)$.

Definition 1. A *system* $\mathcal{S} = \{S_1, \ldots, S_n\}$ is a set of *unit specifications* (also called *specifications*) such that, for all $1 \leq i \leq n$, the *specification* S_i is a tuple $(L_i, A_i, \alpha_i, D_i)$ where L_i is the *language* used to define S_i, $A_i = (s^i_1, \ldots, s^i_{m_i})$ denotes the tuple of *services* of S_i, and $\alpha_i \subseteq \{1, \ldots, n\}$ denotes the set of *indexes of specifications below* S_i. Besides, $D_i = (e_1, \ldots, e_m)$ denotes the tuple of *service definitions* of S_i, that is, each service $s_k \in A_i$ is defined by an expression $e_k ::= f_k(s^i_1, \ldots, s^i_{m_i}, s'_1, \ldots, s'_{m_k})$ where $\{s'_1, \ldots, s'_{m_k}\} \subseteq \bigcup_{j \in \alpha_i} A_j$. This expression is given in language L_i and may depend on any other service of S_i or S_j, with $j \in \alpha_i$. We assume that $j \in \alpha_i$ iff there exists $e_k ::= f_k(s^i_1, \ldots, s^i_{m_i}, s'_1, \ldots, s'_{m_k}) \in D_i$ such that $\{s'_1, \ldots, s'_{m_k}\} \cap A_j \neq \emptyset$. If the service s_k is *atomic* (i.e. it is not defined in terms of other services), we denote it by setting $e_k ::= \llcorner\lrcorner$.

The *set of sub-services* of S_i, denoted by $\texttt{Subservices}(S_i)$, is defined as $\bigcup_{j \in \alpha_i} A_j$. The *specifications of* $\mathcal{S}$ *at level* h, denoted by $\texttt{Level}(\mathcal{S}, h)$, are recursively defined as follows:

$$\texttt{Level}(\mathcal{S}, h) = \begin{cases} \{S_i \mid \alpha_i = \emptyset\} & \text{if } h = 1 \\ \left\{ S_i \;\middle|\; \begin{array}{l} \nexists\, j < h : S_i \in \texttt{Level}(\mathcal{S}, j) \wedge \\ \forall\, v \in \alpha_i \;\exists\, l < h : S_v \in \texttt{Level}(\mathcal{S}, l) \end{array} \right\} & \text{otherwise} \end{cases}$$

If $\alpha_i = \emptyset$ then we say that S_i is *simple*. For a given specification language L, we denote by $\texttt{Units}_L$ the set of all unit specifications in language L. We denote the set of all simple unit specifications in language L by $\texttt{Units}^{\emptyset}_L$. □

Each specification in a system defines a different *unit*. The level of a specification denotes a measure of its dependence on other specifications. Level 1 denotes simple specifications and level 2 denotes specifications depending only on simple specifications. Level 3 denotes specifications depending only on specifications at level 2 as well as those depending on levels 1 and 2, and so on. Let us note that the concept of level applies only if there do not exist *circular dependencies* of

specifications in the system. In this section, we will assume that such dependencies do not appear in our systems. In Section 4 we will remove this restriction. Let us remark that neither individual specifications nor levels correspond with the classical notion of *component*. A unit specification denotes some functionalities of *interaction* of the system with the environment, where each of them may be defined in terms of operations belonging to lower levels. In our framework, we call these functionalities *services*. On the contrary, functionalities provided by a component do no need to interact with the environment. This difference will be relevant for testing purposes. For the sake of simplicity we will assume that service names in a system are *unique*: Given two specifications $S, S' \in \mathcal{S}$ with $S = (L, A, \alpha, D)$ and $S' = (L', A', \alpha', D')$, we have $A \cap A' = \emptyset$.

Example 1. We define a (small) part of the behavior of a client application in a client-server system. Let $\mathcal{S} = \{S_1, S_2, S_3\}$ be a system. The specification S_1 defines how the user makes a request to the server. S_2 defines how to collect from the user the information required by the request, while S_3 defines how to send encrypted/normal messages through a communication channel. These units are defined as follows:

$$\begin{aligned}
S_1 &= (L_1, (request, click, confirmRequest), \{2, 3\}, D_1)\\
S_2 &= (L_2, (userGivesBankData, userGivesName, userGivesAccount,\\
&\quad askForName, askForBankData), \{3\}, D_2)\\
S_3 &= (L_3, (bankReplies, serverReplies,\\
&\quad askBank, askServer, sendName, sendEncryptedAccount), \emptyset, D_3)
\end{aligned}$$

where L_1, L_2, L_3 are (different) specification languages where the sequential execution of operations $a_1, \ldots, a_n$ is expressed as follows: $a_1; \ldots; a_n$. For the sake of simplicity, only sequential executions are considered in this example (e.g., we do not consider *if* statements, *loops*, etc). Besides, D_1, D_2, and D_3 define the services $request$, $userGivesBankData$, and $askBank$, respectively, as follows:

$$\begin{aligned}
request &::= click; askForName; userGivesName; askForBankData;\\
&\quad\ userGivesBankData; askServer; serverReplies;\\
&\quad\ confirmRequest\\
userGivesBankData &::= userGivesAccount; askBank; bankReplies\\
askBank &::= sendName; bankAcknowledgesName;\\
&\quad\ sendEncryptedAccount
\end{aligned}$$

The rest of services are atomic, that is, they are defined as $\llcorner\lrcorner$ by the corresponding D_i (e.g., $click, confirmRequest ::= \llcorner\lrcorner \in D_1$). We will assume that the environment can trigger any service by directly *invoking* it.[2] The action of *calling* a service will be denoted by an auxiliary service. For all service x, we assume that the service $\overline{x}$ invokes its execution, that is, we have the expression $\overline{x} ::= x$. For example, we assume that we have $\overline{confirmRequest} ::= confirmRequest \in D_1$.

The subservices of S_1 are all the services of S_2 and S_3, while $\texttt{Subservices}(S_2)$ consists of all the services belonging to S_3. In addition, $\texttt{Level}(\mathcal{S}, i) = \{S_i\}$ for $1 \leq i \leq 3$. The specification S_3 is *simple* but S_1 and S_2 are not. □

[2] Similarly, given the interface of an object in *object oriented programming*, a user can invoke any object method even if the implementation of the object is a black-box.

Next we introduce a general notion of *test suite*. This concept will allow us to abstract the underlying test derivation methodology. We only assume that there exists a fix criteria to construct test suites (see [8] for a survey on coverage criteria). Though the purpose of the following definition is to generalize known test derivation algorithms, where specifications do not have multiple *levels*, tests are defined in such a way that both simple and multi-level tests can be defined. Simple tests, i.e. tests designed to check a specific unit of the specification as if there did not exist any other units, can be composed to create multi-level tests to test multi-level specifications (this will be described in forthcoming Definition 4). Since tests are designed to interact with the IUT, services activated by a test must be services or subservices of the specification. Regarding the latter ones, let us note that tests are not prepared *by their own* to activate subservices. Hence, if a test activates a subservice then we will denote it by defining it as an atomic service of the test (meaning that a the test does not *know* how to produce it) or by using a service taken from a *lower* level of the test (meaning that the test is multi-level). In the next definition, $\mathcal{P}(X)$ denotes the powerset of the set X.

Definition 2. Let L be a specification language and $S = (L, A, \alpha, D) \in \texttt{Units}_L$ be a specification. A *test* for S is a tuple $T = (L', A', \alpha', D') \in \texttt{Units}_{L'}$ such that $A' = A'' \cup \{fail\}$, where $A'' \subseteq \{x, \overline{x} | x \in A \cup \texttt{Subservices}(S)\}$. Besides, for all $x \in A \cap \texttt{Subservices}(S)$ we have $x ::= \llcorner\lrcorner \in D'$. We denote the set of all tests for S by $\texttt{Tests}_S$. We say that the test $T \in \texttt{Tests}_S$ is *simple* if $\alpha' = \emptyset$ (that is, it stimulates only services belonging to A'). We denote the set of simple tests for S by $\texttt{Tests}^{\emptyset}_S$. A *simple test suite* for the specification S is any element in $\mathcal{P}(\texttt{Tests}^{\emptyset}_S)$. □

We assume that the special service $fail$ is produced by a test when it detects a failure in the IUT according to some criterion. Besides, let us note that the languages used to define specifications and those used to define tests are not necessarily the same.

Example 2. Let L'_1 be a language such that ; denotes a sequential execution and + represents a bifurcation that depends on the next service. We consider the following service, expressed in L'_1:

$$\begin{aligned} askBank ::= \overline{askBank}; sendName; bankAcknowledgesName; \\ (sendEncryptedAccount) + (sendName; fail) \end{aligned}$$

This service defines a test case to interact with the IUT, specifically to check whether the $askBank$ service of the IUT behaves as defined by S_1. First, the test produces $\overline{askBank}$. If the IUT is correct, this service will launch the execution of its service $askBank$. Moreover, if this service is implemented by the IUT as S_1 defines, the IUT will reply with $sendName$. Next, the test will stimulate the IUT with $bankAcknowledgesName$. At this point, the test considers two possibilities. If the IUT replies with $sendEncryptedAccount$ then the test finishes correctly because this is the expected behavior. However, if the IUT produces $sendName$ then the IUT behavior does not conform to S_1, thus leading to $fail$. Let us

note that other definitions of $askBank$ would allow to check other parts of the behavior of this IUT service (e.g., the reply after $\overline{askBank}$).

Let T_1 be a test for S_1 where the previous definition of service $askBank$ is used and the rest of services are defined as *atomic*. T_1 is *simple*. Let us note that T_1 does not need to depend on other tests because the services it uses do not need to be further defined: If the IUT is correct then its implementation of S_1 (i.e. the lowest level) will understand these messages.

Now, let us consider the following service definition:

$$\begin{aligned} userGivesBankData ::= \overline{userGivesBankData}; userGivesAccount; \\ (askBank; bankReplies + askServer; fail) \end{aligned}$$

Let D_2' be a tuple of services definitions including the previous one as well as $askBank ::= \llcorner\lrcorner$, $askServer ::= \llcorner\lrcorner$, and $bankReplies ::= \llcorner\lrcorner$, and let T_2 be a test for S_2 where

$$\begin{aligned} T_2 = (L_2', (userGivesBankData, \overline{userGivesBankData}, \\ userGivesAccount, askBank, askServer, bankReplies, fail), \emptyset, D_2') \end{aligned}$$

T_2 is also a simple test. However, T_2 is useless to stimulate the IUT (at least by itself) because it does not *know* how to produce e.g. $bankReplies$ in such a way that it is *understood* by the IUT. □

In the following definition we introduce a general testing framework. As usual in formal testing, we consider that there exists a formal language to construct a precise model to describe the behavior of the IUT.

Definition 3. Let L be a specification language, $S \in \texttt{Units}_L$ be a specification, and $T \in \texttt{Tests}_S$ be a test. If the interaction of S and T may trigger the execution of a service a then we denote it by $\texttt{Produce}(S, T, a)$. Let $S_1 \in \texttt{Units}_L^{\emptyset}$ be a simple specification, $I_1 \in \texttt{Units}_{L'}^{\emptyset}$ be an IUT, and F_1 be a simple test suite for S_1. We say that I_1 *passes* F_1 if, for all $T_1 \in F$, $\texttt{Produce}(I_1, T_1, fail)$ does not hold. □

As we pointed out before, our testing methodology considers that the IUT is a *black box*. So, we cannot assume any internal structure. In particular, when we speak about a given *unit of the IUT*, we mean the implementation in the IUT of some services that are *logically* grouped in the specification as a unit. Similarly, when we talk about *level of the IUT* we mean the *implementation* of the corresponding units in the IUT, that is, a set of sets of services. If the IUT is correct with respect to the specification then these services must be provided by the IUT and they must be correctly implemented, but their physical structure in the IUT is indeed not considered. In order to test the conformance of a given unit with respect to a specification, we will create tests to stimulate the IUT according to some operations used in that unit. However, each of these operations has to be performed according to its specification, which could be defined in a lower unit. Since the IUT is expected to correctly implement all of the units, the test could take and use operations provided in lower levels of the IUT as a way to perform them.

However, the IUT implementation of these units could be *faulty*. Therefore, before we use the operations given in units belonging to a lower level, we will have to *check* their correctness. More precisely, the correctness of the capabilities provided by those units has to be assessed. In order to do that, we will *test* them. Following the same idea, testing the units belonging to a lower level may require to consider the functionalities condensed in an even lower level of the IUT. So, first of all we will have to check the correctness of those units. The same reasoning is repeated until we infer that we need to check the units corresponding to the lowest level. The tests needed to check the correctness of level 1 do not use any lower unit/level. Once we have tested this level of the IUT, we will use its capabilities as part of the activities of the tests that check the units belonging to the immediately higher level, and so on.

Let us remark that using the services provided by a unit of the IUT as part of the activity of a test does not consist in *breaking* this part and connecting it to the test. Since IUTs are black boxes, this cannot be done. Instead, using an IUT unit consists in taking the *whole* IUT and invoking and using *only* some of its services: The services that are logically grouped as the considered unit in the specification. The next definition formalizes this process. We derive a set of *multi-level* tests from a set of *simple* tests. In tests belonging to the latter set, all services are *atomic*. Thus, they do not need any further definition. In order to obtain the set of multi-level tests, we modify the aforementioned simple tests so that all the tests contain the definition of all lower levels. The operations from these units are taken directly from the IUT. To pass a test suite created for checking the i-th level of the IUT (for some $i > 1$) requires that lower units are correct with respect to the units of the specification defining the same services. In order to be confident in this correctness (although it will not be a *proof* of it), we will recursively apply a suitable test suite to the immediately lower units/level of the IUT. If this test suite is passed then we will use the services appearing in these units to construct services of the tests that check the capabilities corresponding to units appearing at level i.

Definition 4. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ and $IUT = \{I_1, \ldots, I_n\}$ be two systems. Let $S = (L_S, A_S, \alpha_S, D_S) \in \mathcal{S}$ and $I = (L_I, A_I, \alpha_I, D_I) \in IUT$. Let $B = \texttt{Subservices}(I)$. Let $F = \{(L_1, A_1, \emptyset, D_1), \ldots, (L_n, A_n, \emptyset, D_n)\}$ be a simple test suite for S. We say that the set $\{(L_1, A_1 \backslash B, \alpha_I, D'_1), \ldots, (L_n, A_n \backslash B, \alpha_I, D'_n)\}$ is a *multi-level test suite* for S and I, where for all $1 \leq i \leq n$ we have that D'_i is constructed by removing any subservice definition (i.e., $e ::= r$ for some $e \in B$ and r) from D_i. Let G be a multi-level test suite for S and I. We say that I *passes* G for S if the following two conditions hold:

(1) For all S_k, with $k \in \alpha_S$, there exists $l \in \alpha_I$ such that I_l passes G' for S_k, where G' is a multi-level test suite for S_k and I_l.
(2) For all $T \in G$ we have that $\texttt{Produce}(I, T, fail)$ does not hold. □

Let us note that the anchor case of the previous recursive definition corresponds to the case when we test a simple specification. In this case, there is no element

to consider in condition (1), so that this case trivially holds and no recursive call is needed. When a non-simple specification is tested, we recursively test each of its sub-specifications. Afterwards, the whole specification is tested. Since tests belonging to the simple test suite are not provided with definitions of lower level operations, these operations are represented by atomic operations at the actual level. However, after these operations are properly provided by the IUT (tests are linked to α_I, which allows them to use any IUT lower operation), these atomic substitutes are removed. The application of the previous definition directly leads to the iterative testing algorithm presented in Figure 1.

Example 3. Let us consider the test T_2 constructed before. In order to turn this simple test into a multi-level test, we must remove the services $askBank$, $askServer$, and $bankReplies$ from its list of provided services as well as erasing their definition from D'_2 (where they were defined as $\llcorner\lrcorner$), leading to a new tuple of services definitions D''_2. Instead, the new test T'_2 will execute these services by *invoking* their respective implementations at the IUT. In this way, the *implementation* of S_1 in the IUT will enable the test to interact with the *implementation* of S_2 in the IUT. In technical terms, a part of the IUT will be a part the new test. Given a system $\mathcal{S} = \{T'_2, I_1\}$ where I_1 denotes the implementation of S_1 in the IUT, the new multi-level test T'_2 is defined as follows:

$$T'_2 = (L'_2, \{userGivesBankData, \overline{userGivesBankData}, \\ userGivesAccount, fail\}, \{1\}, D''_2)$$ □

Next we show that tests use lower units of the IUT as part of their own definition only after these units have already been tested by previous tests.

Lemma 1. Let $\mathcal{S}, \mathcal{I}$ be two systems such that $S \in \mathcal{S}$, $I \in \mathcal{I}$, and $S \in \texttt{Level}(\mathcal{S}, i)$ for some $i \in \mathbb{N}$. Let G be a multi-level test suite for S and I, and let $T \in G$. The algorithm given in Figure 1 applies T only after, for all $S' \in \texttt{Level}(\mathcal{S}, j)$ with $j < i$, a test T' for S' and some $I' \in \mathcal{I}$ has already been applied. □

4 Specifications with Circular Dependencies

The methodology presented in the previous section is based on the idea that each unit of the specification uses other units that are located *below* it. Hence, we assume the existence of some *minimal* units whose definition does not depend on other units. These units play the role of the *anchor case* in our recursive methodology. This fact makes our methodology to be well-formed. However, one can imagine specifications with *circular* dependencies between units. For instance, let us consider the specification of a distributed system where each part uses services that are provided by all the other parts. In this case, the methodology presented in the previous section might not work because the existence of an anchor case is not guaranteed.

In order to tackle this problem we could consider that all the units containing mutual circular dependencies are part of a *single* logical unit. Hence, the *logical* division of the specification for testing purposes would be free of circular dependencies. However, this method would partially break the modularity of the

Input: A specification S with units grouped in n levels and an implementation I.
Output: A diagnosis result: **true** or **false**.

$i := 1$;
error := **false**;
while $i \leq n$ **and not** error **do**
  Let $\mathcal{S}_i$ be the set of all sub-specifications of S at level i;
  while $\mathcal{S}_i \neq \emptyset$ **and not** error **do**
    Choose $S' \in \mathcal{S}_i$;
    Let I' be the implementation of S' in IUT;
    $\mathcal{S}_i := \mathcal{S}_i \backslash \{S'\}$;
    Generate a test suite G for S';
    if $i \geq 2$ **then**
      forall test $T \in G$ **do**
          use all subparts of I' for implementing level $i-1$ of T
      od
    fi;
    forall test $T \in G$ **do** apply T to I' **od**;
    if *fail* is obtained **then** *error* := **true fi**
  od
  $i := i + 1$;
od
return (**not** error);

Fig. 1. Multi-level testing algorithm

testing procedure. If it were possible, we could also disconnect those units with circular dependencies from the rest of units. Then, we could test each of them in an isolated fashion. Let us note that this requires to *physically* break a part of the IUT. Thus, this alternative is not feasible in a black-box testing approach because we cannot separately access the different parts of the system under test.

In the rest of the section we show another alternative whose main idea is to consider that the order to test units is governed by the *services* instead of by the own units. In fact, there exist several situations where we may have a circular dependence between units but we can still find a sequence of services without a circular dependence between them. If such a condition holds then the order of application of our methodology will be governed by these sequences.

Example 4. A given system consists of two concurrent processes (specified by S_1 and S_2) that can perform dialogs to exchange some information. The service of sending data from one of the processes to the other implies using the service of reception of the last one. We have:

$$\begin{aligned} S_1 &= (L_1, (sendingTo2, receivingFrom2), \{2\}, \\ &\quad (sendingTo2 ::= receivingFrom1, receivingFrom2 ::= \llcorner\lrcorner)) \\ S_2 &= (L_2, (sendingTo1, receivingFrom1), \{1\}, \\ &\quad (sendingTo1 ::= receivingFrom2, receivingFrom1 ::= \llcorner\lrcorner)) \end{aligned}$$

The definition of each of these units depends on the other one, and we have a circular dependence between units. However, the services of reception of data do not depend on any other service. In addition, the services to send data depend on the services of reception, but the dependence of services finishes there. So, there is an order to test services avoiding any circular dependence between them: We test both reception services, and next we check both sending services. That is, in spite of the existence of a circular dependence between units, there is no circular dependence between services. This makes possible to apply our methodology by considering a structure *based on services* instead of one *based on levels*. □

Let us formalize this alternative methodology. First of all, we need to identify all the services conforming a multi-level specification, regardless of the level where the units using them are located.[3] Given a unit, the next recursive function finds all services (as well as their respective definitions) from that unit down through any number of dependence links. The function uses an auxiliary set Q to contain all the services cumulated in previous recursive calls. We will use this set to avoid performing a call over a unit whose services have already been included in the set. By doing so we avoid that a circular dependence produces an infinite sequence of recursive calls to this function. Similarly, we also define the full set of specifications that can be reached from a given specification, through dependency links. In this case, R denotes the set of indexes of specifications already cumulated in previous recursive calls.

Definition 5. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a system and $S = (L, A, \alpha, D) \in \mathcal{S}$ be a specification where $A = (s_1, \ldots, s_b)$ and $D = (e_1, \ldots, e_b)$. Besides, let $P_S = \{(s_1, e_1), \ldots, (s_b, e_b)\}$. We define the *full set of defined services* of S, denoted by $\texttt{Full}(S)$, as $\texttt{Full'}(S, \emptyset)$, where

$$\texttt{Full'}(S, Q) = P_S \cup \left\{ (s, e) \in \texttt{Full'}(S_i, Q \cup A) \;\middle|\; \begin{array}{c} S_i = (L_i, A_i, \alpha_i, D_i) \in \mathcal{S} \\ \wedge\ i \in \alpha\ \wedge\ A_i \not\subseteq Q \end{array} \right\}$$

Let $\texttt{Full}(S) = \{(s_1, e_1), \ldots, (s_m, e_m)\}$. Then, the *full set of services* of S, denoted by $\texttt{FullServ}(S)$, is the set $\{s_1, \ldots, s_m\}$. Let $S_p = (L_p, A_p, \alpha_p, D_p) \in \mathcal{S}$. We define the *set of specifications* below S_p, denoted by $\texttt{Dependents}(S)$, as $\texttt{Dep'}(S_p, \{p\})$, where

$$\texttt{Dep'}(S_p, R) = \{S_k | k \in \alpha_p\} \cup \left\{ S_j \in \texttt{Dep'}(S_i, \alpha_p \cup R) \;\middle|\; \begin{array}{c} S_i = (L_i, A_i, \alpha_i, D_i) \in \mathcal{S} \\ \wedge\ i \in \alpha_p\ \wedge\ \alpha_i \not\subseteq R \end{array} \right\}$$

□

Let us recall that services are defined as an *expression* that may depend on other services. In turn, these services may belong either to the same unit in which the service is defined or to another unit that is directly referred from it.

Example 5. We have $\texttt{FullServ}(S_1) = \texttt{FullServ}(S_2)$, which in turn are equal to $\{sendingTo1, receivingFrom1, sendingTo2, receivingFrom2\}$. We also have $\texttt{Dependents}(S_1) = \texttt{Dependents}(S_2) = \{1, 2\}$. □

[3] The notion of *level* is partially lost in systems with circular dependencies because the maximal distance from some unit to simple units could be *infinite*.

In the next definition we give a condition to construct a sequence containing all the services used in a multi-level specification in such a way that no service is defined in terms of other services that appear *before* in the sequence.

Definition 6. Let S be a specification, $\texttt{Full}(S) = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ be the full set of defined services of S, and $[(s_{i_1}, e_{i_1}), \ldots, (s_{i_n}, e_{i_n})]$ be a permutation of $\texttt{Full}(S)$. The sequence of services $[s_{i_1}, \ldots, s_{i_n}]$ is a *non-circular sequence of services* for S if for all $1 \leq j < k \leq n$ we have that either

- there exists a specification $S' = (L', A', \alpha', D') \in \texttt{Dependents}(S)$ such that $s_{i_j}, s_{i_k} \in A'$, or
- e_{i_k} is not defined in terms of s_{i_j}.

If there exists such a sequence for S then we say that S is *service-structured*. □

If a specification is service-structured then we can develop a methodology where all the services of the specification are tested in a given order: The order given by a list of services fulfilling the condition of the previous definition.

Example 6. $C = [sendingTo1, sendingTo2, receivingFrom1, receivingFrom2]$ is a non-circular sequence of services for S_1 as well as for S_2. □

In the previous section we introduced a testing methodology where test suites for checking the functionalities of each unit were considered. Now we consider tests suited to check a *single* service belonging to the corresponding unit. In order to check the correctness of this service, tests may use only those IUT lower services that appear *further* in the non-circular sequence of services. As we will see, this is valid because these services will be tested *before*. In fact, any other lower service will not appear in the tests constructed to check that service.

Definition 7. Let $S = (L, A, \alpha, D) \in \texttt{Units}_L$ where $A = (s_1, \ldots, s_b)$ and $D = (e_1, \ldots, e_b)$. Let $s = s_k$ for some $1 \leq k \leq b$, and let $e_k ::= f_k(s'_1, \ldots, s'_g, s''_1, \ldots, s''_h)$ where $\{s'_1, \ldots, s'_g\} \subseteq A$ and $\{s''_1, \ldots, s''_h\} \cap A = \emptyset$. Finally, let us consider that $C = [s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n]$ is a non-circular sequence of services of S where $\{s''_1, \ldots, s''_h\} \subseteq \{s_{i+1}, \ldots, s_n\}$, and let $\overline{A} = \{\overline{x} | x \in A\}$. A *test for the specification* S, *the service* s, *and* C is a tuple $T = (L', A', \alpha', D') \in \texttt{Units}_{L'}$ such that

(a) $s \in A'$ *(i.e., s is checked by T)*,
(b) $A' = A'' \cup \{fail\}$ with $A'' \subseteq \{x, \overline{x} | x \in A \cup \texttt{Subservices}(S)\}$, and for all $x \in A \cap \texttt{Subservices}(S)$ we have $x ::= \llcorner\lrcorner \in D'$ *(i.e., T checks only services and subservices of S)*,
(c) $\{fail, s, \overline{s}, s'_1, \ldots, s'_g, s''_1, \ldots, s''_h\} \subseteq \texttt{FullServ}(T)$ *(i.e., T or its sub-levels tackle the full definition of s)* and $\texttt{FullServ}(T) \subseteq \{fail, s\} \cup A \cup \overline{A} \cup \{s_{i+1}, \ldots, s_n\}$ *(i.e., T and its sub-levels do not refer to services before s in the sequence C)*.

We denote the set of all tests for S, s, and C by $\texttt{Tests}_{S,s,C}$. A test $T = (L', A', \alpha', D') \in \texttt{Tests}_{S,s,C}$ is *simple* if $\alpha' = \emptyset$. The set of simple tests for S

is denoted by $\texttt{Tests}^{\emptyset}_{S,s,C}$. A *simple test suite* for the specification S, service s, and list C is any element in $\mathcal{P}(\texttt{Tests}^{\emptyset}_{S,s,C})$. □

Example 7. Next we show a simple test for S_1, $sendingTo2$, and C:

$$T_1 = (L'_1, (sendingTo2, receivingFrom2, recevingFrom1, fail), \emptyset,\\ sendingTo2 ::= \overline{sendingTo2}; (receivingFrom1) + (receivingFrom2; fail),\\ receivingFrom2 ::= \llcorner\lrcorner, receivingFrom1 ::= \llcorner\lrcorner)$$ □

We have the needed machinery to define our *services-oriented* testing methodology. Basically, we will traverse a non-circular sequence of services, checking the correctness of the IUT for each service in the sequence one after each other. For any sequence of services, testing the first service s of the list requires to test before all the services remaining in the sequence. This is done by performing a recursive call where the remaining sequence of services is the parameter.

We will use the services of the IUT already checked in the tests to check other services. We will do it similarly to the way we did it for checking units. In order to check a service used in a certain unit, we create tests that may use lower services, which are provided by lower units of the IUT. However, in the current setting we check a unique service in each step. Hence, it may happen that some of the units that we use as part of the tests are not *completely* tested. Nevertheless, let us note that those services such that the service we are testing depends on them are tested in previous steps indeed. This is so because the order in the sequence of services properly keeps the dependencies between services. Hence, there is no risk to include in a test some functionality that will *actually* be used and has not been checked yet. Next we assume that $[c|C]$ denotes a sequence where the first element c is followed by the sequence C.

Definition 8. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ and $IUT = \{I_1, \ldots, I_n\}$ be two systems. Let $S = (L_S, A_S, \alpha_S, D_S) \in \mathcal{S}$ and $I = (L_I, A_I, \alpha_I, D_I) \in IUT$. Let $B = \texttt{Subservices}(I)$. Let C be a non-circular sequence of services of S. Let $F = \{(L_1, A_1, \emptyset, D_1), \ldots, (L_n, A_n, \emptyset, D_n)\}$ be a simple test suite for S, s, and C. We say that the set $\{(L_1, A_1 \backslash B, \alpha_I, D'_1), \ldots, (L_n, A_n \backslash B, \alpha_I, D'_n)\}$ is a *multi-level test set* for S, I, s, and C, where for all $1 \leq i \leq n$ we have that D'_i is constructed by removing any subservice definition (i.e., $e ::= r$ for some $e \in B$ and r) from D_i.

Let $C = [s|C']$. Besides, let $I' = (L'_I, A'_I, \alpha'_I, D'_I)$ be the unique element in $\texttt{Dependents}(I)$ such that $s \in A'_I$, and $S' = (L'_S, A'_S, \alpha'_S, D'_S)$ be the unique element in $\texttt{Dependents}(S)$ such that $s \in A'_S$. Let G be a multi-level test suite for S', I', s, C. We say that I *passes* C for S if the following conditions hold:

(1) I passes C' for S, and
(2) for all $T' \in G$ we have that $\texttt{Produce}(I', T', fail)$ does not hold.

Besides, we consider that I *passes* [ ] for S, that is, the empty sequence of services is always successfully passed. We say that I *passes* S if I passes C for S, where C is a non-circular sequence of services for S. □

Example 8. Let us suppose that I_1 and I_2 represent the implementation of S_1 and S_2 in the IUT, respectively. Given the system $\mathcal{S} = \{T_1', I_2\}$, T_1' is the multi-level test for S_1, I_1, $sendingTo2$, and C defined as follows:

$$T_1' = (L_1', (sendingTo2, receivingFrom2, fail), \{2\},$$
$$sendingTo2 ::= \overline{sendingTo2}; (receivingFrom1) + (receivingFrom2; fail),$$
$$receivingFrom2 ::= \llcorner\lrcorner)$$

□

Next we show that tests do not use IUT services as part of their own definition until these services have already been tested by other tests.

Lemma 2. Let $\mathcal{S}, \mathcal{I}$ be two systems such that $S = (L, A, \alpha, D) \in \mathcal{S}$, and let $I \in \mathcal{I}$. Let $C = [s_k|C']$ be a non-circular sequence of services of S such that $s_k \in A$ is a service defined by the expression $e_k ::= f_k(s_1', \ldots, s_g', s_1'', \ldots, s_h'') \in D$, where $s_1', \ldots, s_g' \in A$ and $s_1'', \ldots, s_h'' \notin A$. Let G be a multi-level test suite for S, I, s_k, and C, and let $T \in G$. The testing method given in Definition 8 applies T only after, for all $1 \leq j \leq h$, a test T' for some $S' \in \mathcal{S}$, $I' \in \mathcal{I}$, s_j'', and a suffix of C has already been applied. □

5 Conclusions and Future Work

In this paper we have presented a testing methodology that is suitable for testing complex and heterogeneous systems. In these systems, each component can be specified by using a different specification language. These languages can be, in general, very different. The proposed methodology is defined in a recursive way and is based on the idea of testing at first lower levels and by continuing with higher levels, up to the highest one. Testing the correctness of the functionalities of each unit of the IUT allows us to *use* these operations as part of the tests that will check the behavior of units located in higher levels of the IUT. We propose a second methodology allowing to test systems presenting circular dependencies.

References

1. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
2. ISO/IEC 9646-1: 1994. Information technology – Open Systems Interconnection – Conformance testing methodology and framework. Part 1: General concepts (1994)
3. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
4. Núñez, M., Rodríguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005)
5. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)

6. Rodríguez, I., Merayo, M.G., Núñez, M.: A logic for assessing sets of heterogeneous testing hypotheses. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 39–54. Springer, Heidelberg (2006)
7. Rodríguez, I., Merayo, M.G., Núñez, M.: HOTL: Hypotheses and Observations Testing Logic. Journal of Logic and Algebraic Programming (in press, 2007), Available at http://dx.doi.org/10.1016/j.jlap.2007.01.001
8. Zhu, H., Hall, P.A.V, May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surverys 29(4), 366–427 (1997)