# A FORMAL METHODOLOGY TO TEST COMPLEX EMBEDDED SYSTEMS: APPLICATION TO INTERACTIVE DRIVING SYSTEM *

Manuel Núñez,[1] Fernando L. Pelayo[2] and Ismael Rodríguez[1]

[1] *Departamento Sistemas Informáticos y Programación, Facultad de Informática*
*Universidad Complutense de Madrid, 28040 Madrid, Spain*
{mn,isrodrig}@sip.ucm.es

[2] *Departamento de Informática, Facultad de Informática*
*Universidad Castilla-La Mancha, 02071 Albacete, Spain*
fpelayo@info-ab.uclm.es

**Abstract** Complex embedded systems may integrate heterogeneous components. Each component can be defined by using a different specification formalism, or in terms of other components. In order to test the conformance of an implementation with respect to a specification we can use a different testing methodology for each of the (different group of) components. Still, this approach might overlook details regarding the relation among the units of the system under test. In our framework, a *unit* is a set of functionalities of interaction of the system with the environment. These functionalities can be possibly defined in terms of the ones appearing in *lower* units. Essentially, in order to test the functionalities corresponding to a unit, tests may use capabilities described in lower units. These capabilities will be provided by the system under test once they have been previously tested by using some lower-level tests. In order to illustrate our framework we present a running example where the *Interactive Driving System* developed by ADAM Opel A.G. [Opel, 2005] is described.

## 1. Introduction

In order to ensure a greater reliability of critical systems, design languages must be based on a (semi-)formal framework. Actually, in order to build complex systems, it is very important to provide a formal design to contrast the capabilities provided by the system with the expected ones. Depending on the system to be designed, very different specification formalisms can be used.

*Research partially supported by the Spanish MCYT project TIC2003-07848-C02, the Junta de Castilla-La Mancha project PAC-03-001, and the Marie Curie project MRTN-CT-2003-505121/TAROT.

Embedded systems are nowadays so complex that to completely specify their behavior is a very difficult task. In particular, these 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. For each of the parts we may choose the most suitable formalism. Let us remark that the use of different notations is not restricted to the description of different components: It can be used to provide different *views* or interpretations of the same component. This approach has being used in several programming methodologies such as *functional programming* (e.g. [Bird and Wadler, 1988]), *aspect oriented programming* (e.g. [Kiczales et al., 1997]) and *component software* (e.g. [Szyperski, 1998]). In this paper, however, we consider different formalisms only as a means to define different components, not the same component in different ways.

In addition to be composed by several parts, embedded 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. Before we begin to describe how our testing approach is implemented, we will introduce a running example of a *multi-level* specification in the automotive industry: the *Interactive Driving System*, in short IDS, developed by Opel/Vauxhall is the brain of some new models of GM-Europe like Vectra and Astra. What makes this system revolutionary is that it enables the suspension, brake, engine, and steering systems to *talk* to each other in order to provide the best response for different driving scenarios. It senses the driving condition and synchronizes the performance of every component of the chassis and engine to adapt and respond. This behavior results in one of the best coordinated driving dynamics in their respective classes. Simply put, the IDS consists of the following three entities:

- *Electro-Hydraulic Power Steering*, in short EHPS,
- *Electronic Stability Program*, in short ESP, and
- *Shock absorbers Control*, in short CDC.

In turn, we can distinguish the following components in the ESP:

- *Anti-lock Braking System*, in short ABS,
- *Brake Assist System*, in short BAS,
- *Cornering Brake Control*, in short CBC,
- *Electronic Brake-force Distribution*, in short EBD, and
- *Traction Control*, in short TC.

These entities can be seen as *agents* whose behavior consist in performing their work on their particular domains fulfilling the requirements under which they where designed. In order to specify these agents we may consider three different levels of abstraction. The lowest level controls that the hardware components both capture the physical magnitudes and execute the physical orders properly. In order to formally specify the behavior at this level we can use formalisms such as LOTOS [LOTOS, 1988] or SDL [ITU92, 1992]. The intermediate level must guarantee that the real behavior of the vehicle verifies its main objective, that is, anti-lock braking systems guarantee the lock-freeness of the wheels for speeds over $5$ *Km/h*. The behavior in this level can be best defined by using a generic language for the description of agents such as AgentSpeak(L) [Rao, 1996] or 3APL [Hindriks et al., 1998]. The highest level of abstraction is devoted to satisfy the intentions of the driver. Therefore, a general purpose specification language such as UML [Booch et al., 1998] could be an adequate formalism to specify the behavior of agents to control the *speed*, *trajectory*, and *angular moment* of the automobile.

The previous running example shows that the formal specification of complex heterogeneous embedded systems represents an important challenge for traditional testing methodologies. First, specifications are not described by using a unique language. In fact, several languages can be used to specify different (sets of) functionalities, that is, different units. Our notion of unit differs from the concept of *component* where functionalities may be for internal use and do not have to interact with the environment. Systems have a hierarchical structure (different levels). 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. Let us remark that our approach contains, as a particular case, standard approaches for the specification and testing of component-based systems.

Even though the specification of multi-level systems is a stand-alone goal, in this paper we will go one step further. Our goal is to test the conformance of an implementation under test, in short IUT, with respect to a specification. Obviously, even if a system is specified in different stages, 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. For example, in our running example we need to fix which basic messages have to be exchanged between the different units to communicate the current trajectory of the automobile.

An obvious alternative to our approach can be taken 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 it is a black box. Besides, we lose the capability to check the correctness of the *integration* and *assembling* of the parts of the IUT.

The rest of the paper is structured as follows. The next section represents the bulk of the paper. First, we informally present the main features of our testing approach. Next, we introduce the main definitions that will be used during this paper. Finally, we formally define our testing methodology. In Section 3 we introduce how our model is applied to the framework of *Interactive Dynamical Behavior of Automobiles*. Finally, in Section 4 we present our conclusions.

## 2. The multi-level testing methodology

In this section we present our multi-level testing approach. We begin by informally introducing the main characteristics of our approach. We will also review how traditional formal testing works and we will show how these techniques can be adapted/modified/discarded in the case of multi-level testing.

### 2.1 Informal presentation

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 correctness (usually, the incorrectness) of the IUT. These tests usually represent *stimulation plans* of the IUT by providing sequences of events (called *inputs*) that can be interpreted by the IUT. The IUT replies, for each input, with another event (called *outputs*). If the specification language specializes in defining simple communication events 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 inputs to the IUT.

Coming back to our running example, let us suppose that we specify the Interactive Dynamical Behavior of Automobiles, in short IDBA, by using a language that specializes in the definition of the highest level of abstraction. The resulting model will clearly define, for each situation, the desirable behavior. Thus, the specification will determine for each scenario the optimum (possible) behavior of the automobile in terms of speed, trajectory, and angular moment. However, this high-level specification language is not suitable to represent situations such as how should be corrected the torque transmitted to each wheel or how should the brakes act on each wheel. In such a context, one may wonder how a test can indicate to the IUT that a breaking action on the rear wheels has to be performed under a overturn. If the specification language deals only with high level behavior then the test should not interact with the implementation by following a fixed procedure protocol. This is so because the specification language does not allow to express these details. For example, if we fix the procedure protocol when an overturn appears then we are forcing the IUT to follow that procedure, even though this information is not reflected in the specification. However, it is obvious that the test will be using a specific protocol of desirable behavior. Otherwise, it would not be able to propose different actions 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 system, the system itself must indicate some lower level operations allowing to perform the desired functionalities. Thus, the specification of the whole system must be done in several steps and by (potentially) using different specification languages for each level of abstraction.

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. These 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 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. 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 will *use* and *invoke* IUT operations belonging to lower levels as part of its own definition. Before we use these IUT operations, we need to be confident that they are correctly implemented. Thus, we require that these IUT operations have already been tested by other (lower level) tests. This procedure yields a recursive testing methodology where units belonging to lower levels must be tested before the ones corresponding to higher levels.

## 2.2 Basic definitions

Next 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 can be either an *input* or an *output*, being the main difference who is responsible of its initialization: The *outside world* or the *own system*, respectively. 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, an input service can be defined by using an output one, that is in turn defined in terms of an input service, and so on. The organization of services in units allows to precisely define how a unit depends on other units.

DEFINITION 1 Let us suppose that the system contains $n$ different units. For all $1 \leq i \leq n$, a *specification* $S_i$ is a tuple $(L_i, I_i, O_i, \alpha_i)$, where $L_i$ is the *language* to define $S_i$, $I_i = \{s_1, \ldots, s_n\}$ denotes the set of *input services* of $S_i$, $O_i = \{s_{n+1}, \ldots, s_m\}$ denotes the set of *output services* of $S_i$, and $\alpha_i \subseteq \{1, \ldots, n\} - \{i\}$ denotes the set of *specifications below* $S_i$. For all $j \in \alpha_i$, let $S_j = (L_j, \{s^j_1, \ldots, s^j_{a_j}\}, \{s^j_{a_j+1}, \ldots, s^j_{b_j}\}, \alpha_j)$. Each service $s_r$ is defined by an *expression* in the language $L_i$ that may depend on any other service of either $S_i$ or on services defined in $S_k$, for some $k \in \alpha_i$. That is, for any service $s_r \in I_i \cup O_i$ we have

$$s_r = f_r(s_1, \ldots, s_m, s^1_1, \ldots, s^1_{b_1}, \ldots, s^k_1, \ldots, s^k_{b_k})$$

If $\alpha_i = \emptyset$ then we say that the specification $S_i$ is *simple*. We assume that $S_1$ represents the *highest level* specification, that is, for all $1 \leq j \leq n$ we have $1 \notin \alpha_j$ and for any $j \neq 1$ we have that there exists $1 \leq k \leq n$ such that $j \in \alpha_k$.

Let us consider the sub-specifications $S_1, S_2, \ldots, S_n$. The levels of the specification are recursively defined as:

- The first level contains those specifications $S_j$ such that $\alpha_j = \emptyset$.
- For all $i > 1$, the $i$-th level contains those specifications $S_j$ such that for all $k \in \alpha_j$ we have that $S_k$ belongs to an already defined level (that is, a level lower than $i$).
- We finish the process when we find $i$ such that $S_1$ belongs to level $i$.

For a given specification language $L$, we denote by $\texttt{Specs}_L$ the set of all specifications in language $L$. We denote the set of specifications in language $L$ that do *not depend* on any other specification, that is, $\alpha = \emptyset$, by $\texttt{Specs}^{\emptyset}_L$. □

Let us consider the *tree* of sub-specifications included in a specification. Each of these sub-specifications defines a different *unit*. There is a distinguished sub-specification, $S_1$, denoting the *root* of the tree. Specifications at the same level of the tree denote a *level*. Let us remark that neither individual specifications nor levels correspond with the classical notion of *component*.

A sub-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. These functionalities are called *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.

Let us take up again the running example of interactive driving system agents. The highest level of abstraction is given by the behaviors considered acceptable, at each time, by the driver. Essentially, the specification will provide a unique output service "*adequate speed, trajectory within the road, and angular position on the trajectory*." This service will be defined in terms of more basic operations such as "*modify the speed*", normally a reduction, performed by the BAS and ABS agents, "*change the trajectory*," performed by EHPS and ESP agents, and "*angular position tangent to the trajectory*," in charge of CBC and EBD agents. These operations are atomic for the intermediate level of abstraction but they have to be defined in a greater detail for lower levels.

EXAMPLE 2 Let us consider the specification of the output operation "*taking a bend*" at the highest level of abstraction. In order to specify this service in terms of operations of the *interactive driving system* unit, we could consider the following sequence:

- (Output) Turn the steering wheel.
- (Input) Put the foot on the brake pedal.
- (Output) The front wheels turn following the steering wheel and the vehicle decelerates.
- (Input) The driver panics because the new trajectory is not correct. He quickly overturns the steering wheel. In addition, he might step down the brake.
- (Output) As a consequence, the vehicle underturns and follows a right trajectory.
- (Input) Different sensors detect, on the one hand, a violent movement on the steering wheel and, on the other hand, a lost of trajectory since the rotation speed of the front wheels is slower than the one of the rear wheels.
- (Output) An *underturn scenario* from the vehicle is identified.
- (Input) IDS follows the protocol to correct an underturn. This is done by reading the sensors of the agents which is formed of. Once the right trace is selected, within the stored ones, it gives the appropriate instructions

to the braking systems (mainly, braking the wheels of the inside side of the bend, or even also braking both rear wheels via CBC and EBD).

- (Output) The vehicle comes back into the road trajectory (if it is possible from the physics' point of view).
- (Input) IDS returns the control of the brakes to the driver. The vehicle corners in the right way.

The previous dialogue can take different bifurcations depending on the actions on the controls, the speed, the angle of the bend, how slippery the road is, etcetera. □

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 [Zhu et al., 1997] for a good survey on coverage criteria). Since the purpose of the following definition is to generalize current test derivation algorithms, where specifications do not have multiple *levels*, we consider that the given specification is simple. We will extend this notion to multi-level specifications in the forthcoming Definition 5.

DEFINITION 3 Let $L$ be a specification language and $S = (L, I, O, \emptyset) \in \texttt{Specs}_L$ be a specification. We denote the set of tests for the language $L$ by $\texttt{Tests}_L$. We say that a test $T \in \texttt{Tests}_L$ is *simple* if it stimulates only services belonging to $I \cup O$. We denote the set of simple tests for the language $L$ by $\texttt{Tests}_L^{\emptyset}$.

A *simple test suite* for the language $L$ and the specification $S$ is any element belonging to $\mathcal{P}(\texttt{Tests}_L^{\emptyset})$. □

As an additional condition on tests we assume that they provide the output service $fail$ to denote that a failure has been found. 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. In our setting we suppose, again as usually, that specifications and implementations are described in the same formal language.

DEFINITION 4 Let $L$ be a specification language, $S \in \texttt{Specs}_L$ be a specification, and $T \in \texttt{Tests}_L$ be a test. If the interaction of $S$ and $T$ may trigger the execution of a service $a$ then we denote this event by $\texttt{Produce}(L, S, T, a)$.

Let $S \in \texttt{Specs}_L^{\emptyset}$ be a simple specification and $I \in \texttt{Specs}_L^{\emptyset}$ be an IUT. Let $F$ be a simple test suite for $L$ and $S$. We say that $I$ *passes* $F$ if for all $T \in F$ we have that $\texttt{Produce}(L, I, T, fail)$ does not hold. □

## 2.3 Formal definition of the testing process

As we pointed out before, the proposed 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 speak 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 then these services 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. Since the IUT is supposed to correctly implement all of the units, the test should be allowed to take and use operations provided in lower levels.

However, the implementation of the corresponding units conforming these levels 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. Thus, they can be used exactly as they are generated by the corresponding test derivation algorithm for level $1$. Once we have tested this level of the IUT, we will use the contained capabilities as part of the implementation 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 a test implementation 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 corresponding units of the specification. 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 5 Let us consider a specification $S = (L_S, I_S, O_S, \alpha_S)$, an implementation under test $IUT = (L_I, I_I, O_I, \alpha_I)$, and a simple test suite for $L_S$ and $S$, $F = \{(L_S, I_1, O_1, \emptyset), \ldots, (L_S, I_n, O_n, \emptyset)\}$. We say that the set $\{(L_S, I_1, O_1, \alpha_I), \ldots, (L_S, I_n, O_n, \alpha_I)\}$ is a *multi-level test suite* for $L_S$, $S$, and $IUT$.

Let $G$ be a multi-level test derivation suite for $L_S$, $S$, and $IUT$. We say that *$IUT$ passes $G$* for $S$ if the following two conditions hold:

(1) For all $S' = (L', I', O', \alpha') \in \alpha_S$ there exists $i \in \alpha_I$ such that $IUT_i$ passes $G'$ for $S'$, where $G'$ is a test derivation set for $L'$, $S'$, and $IUT_i$.

(2) For all $T \in G$ we have that $\texttt{Produce}(L_S, IUT, T, fail)$ does not hold.

□

Let us note that the anchor case of the previous recursive definition is applied only when we test a simple system. In this situation, there is no element to consider in clause (1), so that this case trivially holds.

## 3. Case study: Interactive Driving System

In this section we present the application of our methodology to our running example: Interactive Driving System Plus by Opel. IDS Plus enables the suspension, brake and steering systems to *talk* to each other. It senses the driving condition and synchronizes the performance of every component of the chassis to adapt and respond. IDS Plus consists mainly of the following three agents:

- *Electro-Hydraulic Power Steering*, in short EHPS, is a remarkable innovation, intelligent enough to sense the *adequate* driving speed. It requires little effort at low speeds (e.g. for parking) and increased effort at high speeds. This ensures safety and complete control.

- *Electronic Stability Program*, in short ESP, basically generates an opposite force to the one which tries to take the vehicle out of the good trajectory. It is composed of the following units:

  - *Anti-Lock Braking System*, in short ABS, guarantees that at any moment the rotation speed of all the wheels are the same and corresponds with the speed of the vehicle during a braking action. It has five sensors to determine those speeds and two electric valves per wheel in order to be able to modify the braking force on any wheel. Its functioning depends on the CBC.
  - *Cornering Brake Control function*, in short CBC, allows to apply braking force individually to each wheel. This helps the car maintain track stability while cornering at high speeds.

  - *Brake Assist System*, in short BAS, recognizes a panic braking situation. In such an emergency, this system releases brake power with a faster built up, considerably reducing braking distance. It acts on the electro valves of the wheels and takes the necessary pressure from an electronic pump which can provide around 150 bar inside the brake circuit.
  - *Electronic Brake-force Distribution*, in short EBD, senses the brake force the driver applies and distributes it proportionally to the mass shift of the car. This helps to improve braking performance even when the car is loaded.
  - *Traction Control*, in short TC, ensures that the vehicle never loses grip of the road, even at high speeds on slippery and wet surfaces.

- *Shock Absorbers Control*, in short CDC, modifies the consistency of the shock absorbers as a function of the angle of the steering wheel and the speed and rock of the vehicle.

These entities can be seen as agents whose behavior consist in performing their work on their particular domains fulfilling the requirements described above. They have to put all their input/output information via a *Controlled Area Network Bus* to the IDS controller. Moreover, the IDS controller, attending more general intentions, will give some new operation instructions to each unit. In order to specify these agents we may consider three different levels of abstraction. At the lower level, every agent among ABS, BAS, CBC, EBD, and TC needs to verify that its input sensors and its hardware mechanisms work properly. At the intermediate level, the agents corresponding to EHPS, ESP, and CDC have to satisfy their own goal, that is, they have to calculate the necessary response in view of the data collected by the sensors. At the highest level, IDS, it has to be ensured that the system follows the intentions of the driver.

Next we briefly describe how our testing approach is applied to this system. Since the IDS controller is the highest level unit, testing this unit implies creating some tests that will stimulate the IUT by proposing some *extreme* situations. In order to allow the resulting tests to communicate with the IUT, we need to endow them with a procedure to allow them to transmit the signals according to the protocol committed by all the agents. Actually, the IUT contains an implementation of the lower-level units that allows it to perform these operations. We will take these units as part of the implementation of each test.

However, the implementation of the lower level units could be faulty. For example, the implementation of the functionalities grouped in the ESP could present mistakes. Hence, before we use it to define our tests we need to check its correctness. In order to do that, we generate a new test suite to check the correctness of the sequences of operations executed as part of a *trajectory correction* in the ESP unit. Tests will be able to perform its operations by interacting through operations belonging to the lowest units of the IUT. These units define how these operations are performed in terms of basic communication operations.

Once we have tested the ABS, CBC, BAS, EBD, and TC units of the IUT we can use its operations as part of the tests that will check the correctness of the ESP unit. If the testing of functionalities belonging to the EHPS, ESP, and CDC units does not find an error then we can use the operations belonging to these units to define tests that will be used to test the highest level unit. Let us remind that by applying these tests to the IUT we will obtain a diagnosis about the correctness of the highest level unit of the IUT and, by extension, about the correctness of the lower units and of the whole system.

## 4. Conclusions

We have presented a testing methodology for testing complex embedded systems. In general, each component is specified by using a different 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. In order to illustrate our approach we have applied it to the interactive driving system developed by Opel.

## References

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.

Booch, G., Rumbaugh, J., and Jacobson, I., editors (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

Hindriks, K., de Boer, F., van der Hoek, W., and Meyer, J.-J. (1998). Formal semantics for an abstract agent programming language. In *Intelligent Agents IV, LNAI 1365*, pages 215–229. Springer.

ITU92 (1992). ITU. Recommendation Z.100: CCITT Specification and Description Language (SDL).

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming, LNCS 1241*, pages 220–242. Springer.

LOTOS (1988). A formal description technique based on the temporal ordering of observational behaviour. IS 8807, TC97/SC21.

Opel (2005). Description of the IDS. http://www.opel.com.

Rao, A. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away, LNAI 1038*, pages 42–55. Springer.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

Zhu, H., Hall, P., and May, J. (1997). Software unit test coverage and adequacy. *ACM Computing Surverys*, 29(4):366–427.