CO-SImulation Trace Analysis (COSITA) tool for vehicle electronic architecture diagnosability analysis

Manel Khlif, M. Shawky, Oussama Tahan

CO-SImulation Trace Analysis (COSITA) Tool for Vehicle Electronic Architecture Diagnosability Analysis

Manel KHLIF, Oussama TAHAN, Mohamed SHAWKY

Heudiasyc-UMR CNRS 6599
Université de Technologie de Compiègne
Centre de Recherches de Royallieu-BP 20529
60205 COMPIEGNE cedex FRANCE

{manel.khlif; oussama.tahan; shawky}@hds.utc.fr

Abstract-**In this paper we present a CO-SImulation Trace Analysis (COSITA) tool in order to analyze functional/architectural properties, in the automotive field. These properties should enhance a specific design requirement that we call functional/architectural diagnosability. The validation process is applied on a real automotive experimental embedded platform called DIAFORE based on several Electronic Control Units.**

In the design phase of a System Development Life Cycle, we aim to analyze the functional/architectural diagnosability by analyzing its properties (observability, reachability, etc.), using a co-simulation-based approach. In this paper, our objective is to verify that the analysis made on the real platform is consistent with the theoretical analysis made on the co-simulation results.

We believe that simulating a functional model is not sufficient to analyze system properties, because each hardware instantiation has its own properties constraints and limitations. Therefore, a hardware architecture characteristics modification may change the system requirements correctness. Hence, we co-simulate the Hardware (HW) and Software (SW) models, and then we analyze the interaction between them, by analyzing the co-simulation trace.

For co-simulation, we use SystemC and Matlab/Simulink tool, with objectively selected simulation scenarios reflecting the system behavior.

Keywords - ***Automotive electronic embedded systems, Hardware-Software co-modeling and co-simulation, Diagnosability analysis.***

I. INTRODUCTION

NOWADAYS, systems in the automotive industry are becoming more complex due to the growing demand on driving assistance functions. In order to improve the reliability of the design, we have to foresee the possibility of system failure. Assessing the diagnosability of a system regarding a defined set of faults ensures that any eventual occurrence of one of the faults will be correctly detected, identified and hence isolated.

In [1], a new framework for design of automotive systems was defined as result of the IDD (Integrating Diagnosis in the Design of automotive systems) European projects, especially Model-Based Diagnosis which is suitable for integrating diagnosis in the design of electronic systems; both from the methodological and the practical point of view [2].

In this work, we consider embedded on-line Model-based Diagnosis [3] as a requirement at the design time. The model-based approaches for diagnosis and diagnosability analysis that were developed in the domains of automatic control and artificial intelligence can be classified into two categories:

- state-based approaches, which are suitable for continuous systems as shown in [4],
- and event-based approaches [5], which are suitable to use with discrete-events models to represent an electronic embedded system [6]. At the system modeling phase in event-based

approaches, we should provide the maximum of faults models.

There is also a recent approach that deals with hybrid systems where both continuous and discrete aspects coexist in modeling [7][8]. However, in all these categories, there is an absence of the architecture description. Hence, the limitations on the diagnosability imposed by the hardware architecture are not considered, therefore we present in this paper the first step that will lead us to analyze, the diagnosability requirement of an embedded hardware architecture, taking into account simultaneously functional (software) and architectural (hardware) aspects.

To present our contributions, this paper is structured as follows:

First, we present the functional/architectural diagnosability analysis. Then in section III, we present the co-design technique that we use for co-simulation. In section IV, we describe our tool COSITA (CO-Simulation Traces Analysis) that we have developed to analyze several properties. In section V, we discuss the hardware architecture properties.

Section VI shows some results. We validate the results via an automotive platform in section VII. Finally, we conclude this paper and present our future works in section VIII.

II. FUNCTIONAL/ARCHITECTURAL DIAGNOSABILITY

A. *Diagnosability Definition*

Our tentative definition would describe Diagnosability as the property of a system allowing it to react face to a set of anticipated faults, in order to reduce the cases where no decision can be made after fault detection.

B. *Scope of the Functional/Architectural Diagnosability Metrics*

We have updated the definition of classic diagnosability metrics and define new ones to adapt them for functional/architectural analysis: Observability and reachability. The different diagnosability metrics that we defined are:

- The observability is defined as a Boolean property, or a time schedule of available windows for observations in a sub-system (I/O, communication bus) or a degree for the whole system,
- The reachability is a Boolean property, representing the possibility for a node to reach an I/O or a variable value;

The diagnosability is often defined as a degree, i.e. a percentage combining the observable and reachable I/Os or variables over their total number.

These metrics will be analyzed through Co-modeling and Co-simulation techniques described in the following section.

III. HW/SW CO-MODELING AND CO-SIMULATION

The electronic HW/SW architecture that we co-model is composed of ECUs (Electronic Control Units) interconnected by a communication bus [9]. Every ECU is composed of a processor, a memory, a CAN interface and I/O interfaces.

A. *SystemC/ Simulink Co-Modeling*

In the first place, we used SystemC as a HW/SW Co-Modeling language for the entire system [10].

Therefore, it is not possible to implement the software part of the model developed in SystemC on our platform (Fig. 7) since this latter supports only C code generated from Simulink Blocks. Furthermore, Simulink is well established in the automotive industry as a quick modeling tool, offering automatic code generation.

Therefore, we use SystemC as modeling language for hardware architecture and Simulink as functional modeling language. So, we need to interface both SystemC and Simulink models as it is done in [11]. We used the «engine.h» C++ library (in the SystemC code), which allows commanding Matlab, stopping it and exchanging data with it from a program written in C, C++, Fortran, etc. Then, Matlab launches Simulink environment for simulation.

B. *ECUs and CAN Bus Modeling*

In this part of the work, we have modeled in SystemC the CAN protocol real-time behavior to realize communications between ECUs models (Fig. 1). We have simplified the details to ease the modeling by implementing a virtual arbiter in the bus. With the Transaction Level Modeling (TLM), the communication between components is described as function calls.

Each ECU that needs to send a message transmits a request to the bus. If at least 2 ECUs request a bus transmission at the same time (i.e. in a time shorter than a bus cycle), the bus arbiter selects the most important message by comparing arbitration fields in the two messages. The same clock is used for all processors as the level of modeling granularity is high.

It is important to note that full CAN protocol is used only in models with high level of granularity, expressing transactions between ECUs.

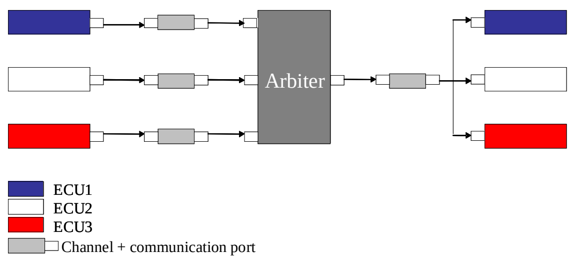


Fig. 1. CAN bus SystemC model

C. SystemC / Simulink Co-Simulation

The co-simulation interface may be in one of two ways: Either SystemC is in charge of controlling the simulation (as SystemC environment includes also a simulator) by running the Simulink model through MATLAB, or Simulink is considered as the master of the simulation and controls SystemC models [12][13].

For our system, we adopted the first approach simply because it is the architecture (ECUs and CAN bus) that includes the embedded function (Fig. 2).

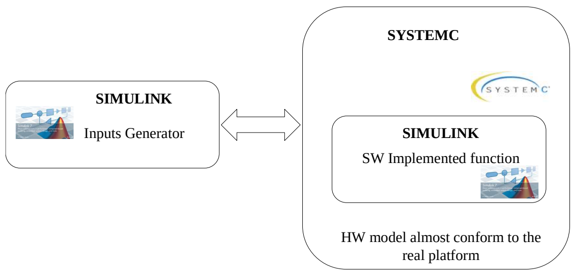


Fig. 2. SystemC-Simulink interfacing

SystemC enables us to properly model the ECUs, the CAN bus and the interconnections. The developed HW model in SystemC will be our virtual platform quite close to our real one.

IV. COSITA Tool

We have developed the COSITA tool in order to have an interface between co-simulation and properties analysis (Fig. 3).

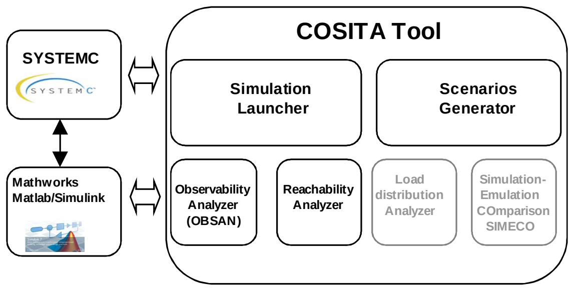

COSITA Tool
SYSTEMC
Mathworks Matlab/Simulink
Simulation Launcher
Scenarios Generator
Observability Analyzer (OBSAN)
Reachability Analyzer
Load distribution Analyzer
Simulation-Emulation COmparison SIMECO

Fig. 3. COSITA modules

A. Simulation Inputs

One of our objectives is to generate scenarios for the execution of the simulated and implemented application. Therefore the first idea is to generate random values for Inputs/Outputs. But we aim to have a more intelligent approach to generate these values. Therefore, we used the dichotomous approach to cover the I/O signal range. On the other hand, several other scenarios generation methods may be used based on condition and decision coverage techniques.

Our tests will then be based on running N simulations, and in each run we apply new selected input values that may be considered as critical for the system.

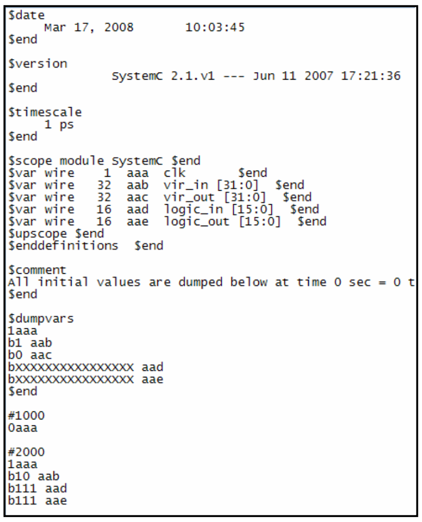
B. Trace Files Generation and Analysis

The SystemC-Simulink co-simulation generates dated log files with all information about values changes of variables, ports…etc. While designing the SystemC model, we should indicate that we need to generate the simulation trace files, by adding the simple instruction “sc_create_vcd_trace_file()” to the program.

Trace generation will be done by writing in a chronogram file; each signal can be logged simultaneously in several files.

These files reflect the activity of the system. Trace files analysis can monitor the total behavior of a hardware architecture. For example, we can trace the internal and external activities of an ECU through its ports identifiers, variables identifiers, etc.

These trace files are in Value Change Dump (VCD) format. This VCD format is specified in the standard IEEE 1364. The VCD file starts with header information giving the date, the simulator’s version number and the timescale used. Next, the file contains definitions of the scope and type of variables being dumped, followed by the actual value changes at each simulation time increment. Only the variables that change value during a time increment are listed. The simulation time recorded in VCD file is the absolute value of the simulation time for the changes in variable values (Fig. 4).



```
$date
     Mar 17, 2008       10:03:45
$end

$version
              SystemC 2.1.v1 --- Jun 11 2007 17:21:36
$end

$timescale
     1 ps
$end

$scope module SystemC $end
$var wire    1  aaa  clk       $end
$var wire   32  aab  vir_in [31:0]  $end
$var wire   32  aac  vir_out [31:0]  $end
$var wire   16  aad  logic_in [15:0]  $end
$var wire   16  aae  logic_out [15:0]  $end
$upscope $end
$enddefinitions  $end

$comment
All initial values are dumped below at time 0 sec = 0 t
$end

$dumpvars
1aaa
b1 aab
b0 aac
bxxxxxxxxxxxxxxxx aad
bxxxxxxxxxxxxxxxx aae
$end

#1000
0aaa

#2000
1aaa
b10 aab
b111 aad
b111 aae
```

Fig. 4. VCD file format

The tool suite COSITA launches SystemC-Simulink co-simulation by generating selected scenarios for co-simulation. After that, COSITA collects co-simulation traces files, merges them and analyze functional/architectural diagnosability metrics (observability and reachability) by parsing these trace files.

In a similar way of formal methods property checking techniques, we use the analysis of simulation traces method to verify certain HW-SW properties of the system modeled in SystemC-Simulink. The analysis of simulation traces has been discussed and compared to the SystemC-Simulink code parsing possibility, before it is applied.

V. FUNCTIONAL/ARCHITECTURAL DIAGNOSABILITY PROPERTIES ANALYSIS

Setting HW-SW properties for analyzing electronic architecture is strongly linked to the definition of requirements at the beginning of the development cycle. Thus, to satisfy the requirement of diagnosability in automotive architecture, we defined, but not limited to, two properties to be analyzed; Observability and Reachability.

A. Observability Checking

By observability, we mean the possibility for different functions of a system (identified by their I/Os and memory variables) to be observed by an independent diagnosis process, without interfering with the nominal operation of the system. Therefore, we check the observability potential (property) for an electronic system to allow adding a process of real time observation of the system's behavior.

We consider the observability property as a Boolean indicator for the functional/architectural diagnosis ability. It is true when the time available is sufficient for on observing process to carry out the periodic surveillance process. Hence, to determine the available time intervals for the observation process, we seek the free cycles not exploited by the nominal operation of the system.

After that, we compare the duration of the available cycles with the time necessary to execute the observation process using the same hardware resources. Keeping the same principle of analysis, this property could be also represented as an observation schedule. However, if the architecture becomes complicated, observability is represented as a degree or a Boolean as we cannot combine different observation schedules. This subject has been discussed in an earlier paper [14].

B. Reachability Checking

We define reachability property as a Boolean indicator for the diagnosability of the hardware architecture. It is true when an ECUi can get access to Inputs/Outputs values of an ECUj, when both ECUi and ECUj are connected through the communication bus.

To determine the reachability of an ECUi to an Input/Output proper to ECUj, we seek messages sent from ECUi, having as a target the Input/Output in ECUj, by analyzing communication traces.
For example in the (Fig. 5), we have S3=f(E1).To get the E1 value to compute f(E1), E1 of ECUj should be reachable from ECUi.

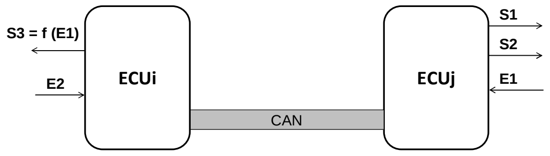


Fig. 5. ECU Input/Output reachability

VI. PROPERTIES ANALYSIS RESULTS

We have tested our approach on the Smart Distance Keeping (SDK) function, given by a truck manufacturer. "SDK" is equivalent to the Adaptive Cruise Control (ACC) function, except that the distance/speed regulation is based only on a fixed distance of 50 m (compliant to European regulations for heavy trucks) [15]. Thus, using embedded radar, the SDK sub-system maintains a safe headway time, i.e. the inter-vehicle distance is varying as a function of the velocity and is maintained at a minimum legal distance of 50 m (Fig. 6).

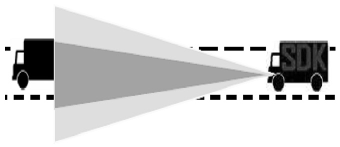


Fig. 6. SDK (Smart Distance Keeping) function

TABLE 1 shows that 40% of the free cycles can execute observation process on the CAN bus and 30% on the global system (ECUs and CAN bus).

The Boolean value “1” for the reachability of “Relative distance” (Input) and Vehicle speed (Output) confirm the observability of the event.

TABLE 1
Example of Functional/Architectural Properties Analysis Values

Property	Global system	New distance	Relative distance	Vehicle speed	CAN bus
Observability	30%	0			40%
Reachability			1	1	

VII. ANALYSIS VALIDATION

A. *DIAFORE: The emulation Platform*

Simulation helps engineers validate and to have a viewpoint of hardware architecture design. However, we are not able to test the physical real world behavior of embedded functions through simulation. Alongside the simulation, emulation allows the validation team to make tests very close to the final operation on the final electronic system, but on a unique configuration preset by the manufacturer of the testing platform.

DIAFORE is an automotive embedded platform composed of several Electronic Control Units from Continental Corporation and based on Motorola microcontrollers. These units can communicate with each other, with a development PC and with other devices through the CAN bus (Fig. 7).

To implement an application on the platform, the first step is to create a Simulink model of the application using blocks with a given sample time. When the application is validated during the simulation phase, we add Motohawk Simulink blocks to specify the physical parameters of the ECUs (Inputs/Outputs, CAN configuration, Triggers period etc.) [16][17][18]. Once the system and the application are ready, we generate the corresponding C code using the RTW (Real Time Workshop) of Simulink and Greenhills software [19].

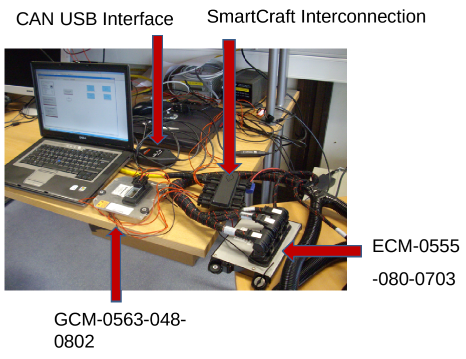


Fig. 7. DIAFORE platform interconnected components

B. *Comparing Simulation-Emulation Results*

In order to validate the results of an architecture properties analysis, we check properties not only for simulated models but also for developed applications running on a real automotive hardware architecture. Then we compare both simulation and emulation results.

All simulated models are not 100% accurate compared to real devices due to discrepancies in sampling rates or to other physical constraints.

In order to compute and compare the different results, we generate trace files from both simulation and emulation. Traces are obtained under two different formats depending of the source. Simulation results trace is obtained directly under a VCD file format while emulation results trace is obtained under log files format.

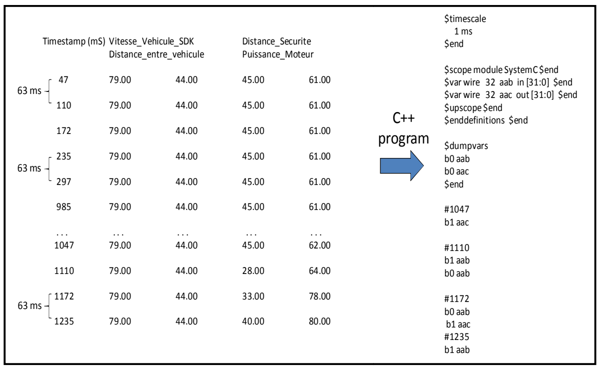
- **Trace file generation from emulation**
  The generation of these files is done by using CANalyzer tool from VECTOR. CANalyzer is a software tool that allows users to analyze networks and distributed embedded systems [19]. We used it to communicate with the ECUs through the CAN interface. At each Input/Output access, and each read or write of the variables in the ECU’s memory, the Electronic Control Unit sends a specific CAN packet on the communication bus (CAN bus) containing the

value of Inputs/Outputs or variables, while CANalyzer collects these special packets. Each packet has its own ID (IDentifier) that reflects the ID of the considered Input/Output or variable. Therefore, at the end of the tests, we obtain a text file (log file) that contains the trace of all the values of Inputs/Outputs and variables with their access instants.

We have chosen CANalyzer to create the log file due to its time accuracy, which is 1 ms. So the program implemented in CANalyzer has an internal counter with a resolution of 1 ms. Compared to the simulation tests, it is considered as a restriction since with the simulation process we can obtain a resolution of about picoseconds or microseconds, which is impossible for our real platform tools.

- **Trace file generation from simulation**
  The generation of these files is done by the simulator in the VCD format as described in IV.

In order to unify analysis methods and easily compare simulation and emulation results, we transform the emulation traces files from a log format to a VCD format that contains only the instants where there are values changes (Fig. 8).



```
Timestamp (mS) Vitesse_Vehicule_SDK              Distance_Securite
               Distance_entre_vehicule           Puissance_Moteur

       ┌ 47     79.00       44.00       45.00       61.00
63 ms -┤
       └ 110    79.00       44.00       45.00       61.00

         172    79.00       44.00       45.00       61.00

       ┌ 235    79.00       44.00       45.00       61.00
63 ms -┤
       └ 297    79.00       44.00       45.00       61.00

         985    79.00       44.00       45.00       61.00

         ...     ...         ...         ...         ...
        1047    79.00       44.00       45.00       62.00

        1110    79.00       44.00       28.00       64.00

       ┌1172    79.00       44.00       33.00       78.00
63 ms -┤
       └1235    79.00       44.00       40.00       80.00
```

```
$timescale
   1 ms
$end

$scope module SystemC $end
$var wire  32  aab  in [31:0]  $end
$var wire  32  aac  out [31:0]  $end
$upscope $end
$enddefinitions  $end

$dumpvars
b0 aab
b0 aac
$end

#1047
b1 aac

#1110
b1 aab
b0 aab

#1172
b0 aab
 b1 aac
#1235
b1 aab
```

Fig. 8. Converting log file to VCD file

C. *Diagnosability metrics analyses comparison*

The comparison analysis results issued from simulation and emulation is directly related to SystemC-Simulink co-simulation results and to emulation on the automotive platform. We have to retrieve the generated VCD text files issued from co-simulation and from emulation, analyze them and identify the differences. In the following we present different situations of the analysis comparison:

- The first situation shows exactly the same analysis results in both simulation and emulation. For example, for the SDK embedded function, we may have the same observability analysis result where observability frequency and observable and non observable slots of time are exactly the same in both tests (Fig. 9). This situation is not frequent; it reflects a HW-SW co-model precisely similar to the real platform, which is hard to obtain.

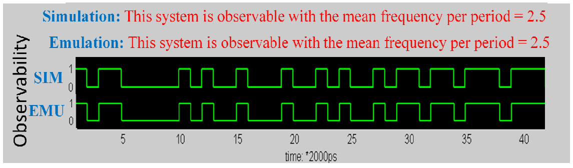


Fig. 9. Observability analyses comparison (First situation)

- The second situation presents a slight discrepancy between the two results; observable slots of time may slightly differ between simulation and emulation (Fig. 10). It may be the most probable situation.

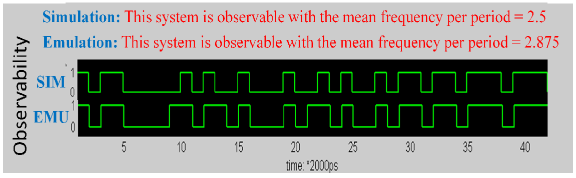


Fig. 10. Observability analyses comparison (Second situation)

- The last situation presents two different cases where a large difference can occur. For instance, concerning the observability analysis, we may have two different scenarios:

  a. Both results are observable with different observable time slots (Fig. 11).

  b. Only one of the two results is observable (Fig. 12).

These two scenarios may be caused by different reasons related to the hardware, because we use the same software model in simulation and emulation, such as:

- Hardware modeling errors introduced in the co-design phase,
- Hardware platform failure or malfunctioning while emulation,
- Large difference between HW-SW distributions used in simulation and emulation.

This situation reflects a divergence between the HW-SW co-model and the real platform and requires a re-examination of both hardware model and real platform hardware from designers in order to check them.

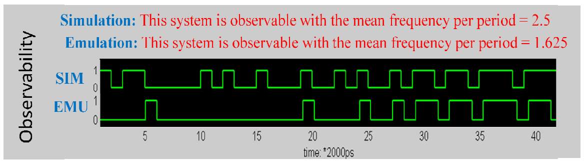

Simulation: This system is observable with the mean frequency per period = 2.5
Emulation: This system is observable with the mean frequency per period = 1.625

Fig. 11. Observability analyses comparison (Third situation, first scenario)

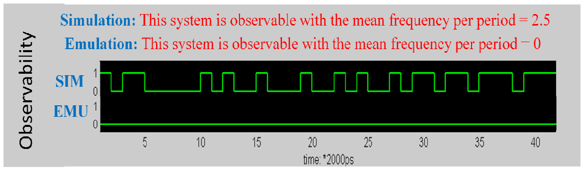

Simulation: This system is observable with the mean frequency per period = 2.5
Emulation: This system is observable with the mean frequency per period = 0

Fig. 12. Observability analyses comparison (Third situation, second scenario)

VIII. CONCLUSIONS AND PERSPECTIVES

This paper presented a tool and an approach to analyze functional/architectural diagnosability properties in automotive electronic architecture. This approach is based on co-simulation trace analysis. We consider this work as a seed tool for the functional/architectural diagnosability analysis methodology.

In our future work we aim to extend COSITA tool to analyze other properties that we are defining in order to fulfill diagnosability analysis and to suggest automatically new configurations of the hardware/software architecture if the system is not able to integrate a diagnosis process. On the other hand, in order to cover most of the different possible simulation scenarios, we aim also to use more advanced methods to generate inputs and variables values taking into account critical situations. In addition, we are currently developing SIMECO (SIMulation/Emulation COmparison) tool in order to automate the comparison of simulation and emulation traces.

IX. ACKNOWLEDGMENTS

The implementation on the physical platform has been done on the equipped vehicles of our laboratory, where the engineering team, including Mr. Dherbomez and Mr. Hanna has been involved.

X. REFERENCES

[1] C. Picardi, R. Bray, F. Cascio, L. Console, P. Dague, O. Dressler, D. Millet, B. Rehfus, P. Struss et C. Vallée, "IDD : Integrating Diagnosis in the Design of automotive systems," *15th European Conference on Artificial Intelligence ECAI-02*, Lyon, juillet 2002

[2] L. Console, and O. Dressier, "Model-based diagnosis in the real world: lessons learned and challenges remaining," in Proc. *16th IJCAI*, pp.1393-1400, Stockholm, 1999.

[3] W. Hamscher, L. Console, and J. deKleer, "Readings in model-based diagnosis," *Morgan Kaufmann Publishers*, ISBN: 1-55860-249-6, San Francisco, CA, USA, 1992.

[4] L. Travé-Massuyès, M, O. Cordier, and X. Pucel, "Comparing diagnosability in cs and des," *In Proceedings of the 6th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, Beijing, China, August 2006.

[5] M. Sampath, R. Sengupta, and S. Lafortune, "Diagnosability of discrete-event systems," *IEEE trans. On Automatic Control 9(40), pp. 1555-1575*, 1995.

[6] F. Lin, "Diagnosability of discrete-events systems and its applications," *Discrete Event Dynamic Systems*, vol. 4, pp. 197-212, may 1994.

[7] G.K. Fourlas, K.J. Kyriakopoulos et N.J. Krikelis, "Diagnosability of hybrid systems", *Proceedings of the 10th IEEE Mediteranean Conference on Control and Automation (MED2002)*, 2002.

[8] M. Beyoudh, L. Travé-Massuyès et X. Olive, "hybrid systems diagnosability by abstracting faulty continuous Dynamics", *Proceedings of the 17th International Workshop on Principles of Diagnosis (DX'06)*, pp. 9-15, 2006.

[9] D. Paret, **"**Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire," *Wiley*, ISBN: 978-0-470-03416-3, 2007

[10] T. Grötker, and al, "System Design with SystemC," *Springer*, Chapter 8, p. 131. ISBN 1402070721, 2002

[11] C. Warwick, "SystemC calls MATLAB," *MATLAB Central*, March 2003, http://www.mathworks.com/matlabcentral/

[12] F. Czerner, and J. Zellmann, "Modeling Cycle-Accurate Hardware with Matlab/Simulink using SystemC," *6th European SystemC Users Group Meeting (ESCUG)*. Stresa, Ilalia, 2002

[13] J-F. Boland, and al, "Using Matlab and Simulink in a SystemC verification Environment," *2nd North American SystemC User's Group*. Santa Clara, CA, USA, 2004

[14] M. Khlif, and M. Shawky, "Observability Checking to Enhance Diagnosis of Real Time Electronic Systems," *DS-RT 2008. The 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications.*October 27 - 29, 2008., Vancouver, Canada, 2008

[15] X. Claeys, and al, "Chauffeur Assistant Functions," *Report restricted to RENAULT TRUCKS*, European Contract number IST-1999-10048, Lyon, FRANCE, 2003

[16] MGM.*Mototron Inc*. ECM-0555-080-0703-F Data Sheet-20/10/2006

[17] MGM.*Mototron Inc*. GCM-0563-048-0802 Data Sheet-14/08/2006

[18] http://www.mototron.com/support/wiki/index.php?title=MotoHawk

[19] http://www.ghs.com/

[20] http://www.vector-france.fr/vf_canalyzer_fr.html