# Data Consistency Testing in Automotive Multi-Core Applications - towards systematic requirement elicitation

Ralph Mader
Lucian Bara
Vitesco Technologies GmbH,
P.O. Box 100943
D-93009 Regensburg
Germany
Email:ralph.mader@continental-corporation.com
Email:lucian.2.bara@continental-corporation.com

Stefan Resmerita
Anton Poelzleitner
Wolfgang Pree
University of Salzburg and Chrona.com
Jakob-Haringer-Str. 2
5020 Salzburg
Austria
Email: stefan.resmerita@cs.uni-salzburg.at
Email: anton.poelzleitner@cs.uni-salzburg.at
Email: wolfgang.pree@chrona.com

***Abstract*—Today multi-core micro-controllers are widely used in automotive applications. Increased performance requirements and the need for speed up increasingly require the distribution of software across cores. Independent of what type of software is used, legacy or AUTOSAR-Classic [1], ensuring data consistency is fundamental for the proper execution of the functionality. For performance reasons data consistency shall not be over-specified, which could result in a too high CPU load and memory allocation due to a massive usage of semaphores. For this purpose, AUTOSAR-Classic provides the possibility to specify consistency requirements for optimizations. But how can the crucial consistency requirements be identified? Today this is mainly done by a design review of the software module with respect to its input variables. In this paper we propose a method which allows the identification of consistency requirements for a module in a formal way. It is based on the identification of so-called problematic access patterns for the input data which are then enforced in the Software In the Loop (SIL) in a dynamic stress test. The results of the module are checked against the expected test vectors. In case such problematic access patterns create a violation of data consistency, that is, a deviation from the expected test result, the generation of a corresponding consistency requirement is indicated to the developer. The method can be integrated into an existing test environment for model based development.**

***Index Terms*—multi-core, real-time system, AUTOSAR, automotive, embedded software, data consistency, verification**

## I. Multi Core Software for Automotive Applications

### A. Where Multi-Core is in use

Since 2015 multi-core micro-controllers are state-of-the-art in various Electronic Controll Units (ECUs) in the automotive domain such as the powertrain domain: engine management-, inverter-, transmission- and domain-controllers are equipped with controllers with up to four cores. The motivation to use multi-core micro-controllers is manifold:

1) like a single core controller, just due to the additional flash and RAM memory
2) for safety reasons, where a second core is used to control the first one in a lockstep mode
3) as integration platform, to reduce the number of controlled units in the vehicle, combining loosely coupled applications
4) for performance reasons, when one core is not sufficient to perform all needed calculations of one application in time

While in the first two use cases no inter-core communication has to be considered the demand increases for the latter two. Engine management systems [2] are a good example for category 4, where a complete application has to be distributed across three or more cores. The coupling of functionality in this application is quite high such that a significant amount of data has to be transferred in a consistent manner. [3].

### B. Legacy and AUTOSAR-Classic Code as basis for distribution

The software which is used varies significantly. For products which have been in place for a long time, a significant amount of legacy code had to be migrated to the multi-core ECUs. Even the migration to AUTOSAR-Classic did not solve every multi-core issue especially in use case 4, if the distribution of software by components is not sufficient to satisfy the performance demand [4]. In any case a consistent data transfer is required. Therefore we had to adapt the software platform for the different categories (see Figure 1). The main difference compared to a standard AUTOSAR-Classic architecture is the so called "Multi Core Layer" for an efficient protection of data with respect to consistency.

### C. Project versus Platform Development of Software

When developing software, the aspects of reuse in different projects have to be considered. While making one project multi-core-capable one would freeze the core distribution and

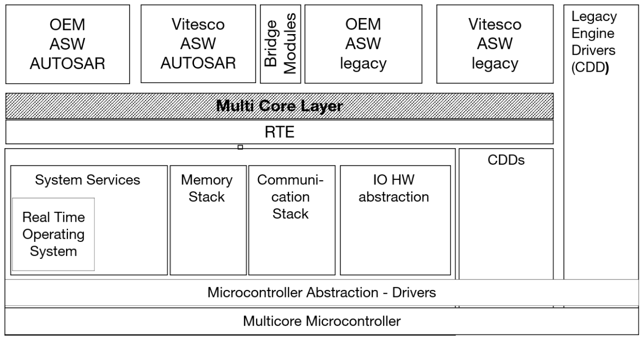


Figure 1. AUTOSAR-Classic based Software Architecture in Powertrain Applications with Multi Core Layer to ensure data consistency. OEM - Original Equipment Manufacturer. CDD - Complex Device Driver. RTE - Runtime Environment

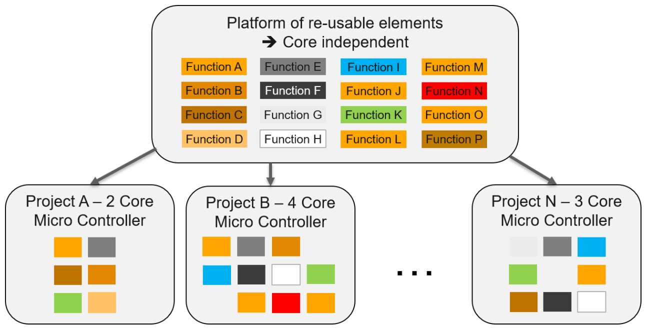


Figure 2. Deriving projects with different Core partitioning out of a generic platform

verify every data transfer. Additional means are required for a software module which shall be integrated in different projects with different core distributions (see Figure 2).
As a software component and the corresponding runnable entities have to be developed in a way that independent of the allocation to a core in the system, the data can be protected if required but allows still freedom to optimize if the protection is not needed, e.g. due to non-concurrent scheduling of the tasks.
As a consequence we have taken the decision to describe requirements for the integration, especially for data consistency, from the consumer perspective assuming always a worst case scenario, that is all data are produced on different cores. The code has been migrated in a way that all accesses to global shared data is done via Get-Set-methods. The default implementation of the Get-Set-methods, which allows the compilation of the module in any environment, is patched in the context of a multi-core project with the appropriate access implementation either to buffered data, to ensure consistency, or to global shared data, in case no protection is required. The calculation of needed buffers and the access modification is performed by a tool developed Continental internally [3]. A similar approach has been applied to AUTOSAR-Classic code. In this case the access to data is done as well via Get-Set-methods which hold as implementation RTE iRead and iWrite implementations. With the help of the so called RTE plug-in concept in AUTOSAR-Classic [5] it is also possible to adapt the access to consistency buffers according to the same algorithm as for the legacy code.

## II. Data Consistency

### A. Stability and Coherency

Data consistency is divided in two categories, as follows.
*Data stability* over time: if a consumer accesses an atomic variable multiple times in a sequence it is expected that the same value is always read. If such data is produced in another context (a high priority task or a task on another core) this cannot be guaranteed in every case.
*Data coherency*: a set of two or more atomic data which are expected to have a certain relationship to each other. One example could be a measurement value and a corresponding flag which indicates the quality of the measurement value.

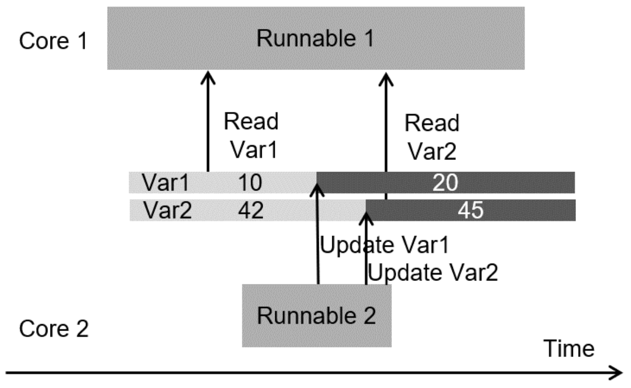


Figure 3. How two scalar data can become incoherent!

Figure 3 shows that the consumer runnable 1 on core 1 accesses variables 1 and 2 and expects a certain relationship between them, which is satisfied, when they have values 10 and 42. Other value combinations might lead to a malfunction. The producer of variable 1 and 2 is executed on core 2 which interferes with the read access of runnable 1 and it might happen that runnable 1 receives an incoherent set of data.

### B. Means to ensure Data Consistency

To ensure data consistency, we have several means to guarantee this in any case:

1) buffering for stability
2) buffering for coherency
3) exclusiveness of access by scheduling

To guarantee stability for atomic data, it is sufficient to create a copy of the content in a buffer memory and let the consumer access the buffered data.
To ensure coherency of atomic data, in addition to the consumer access to buffered data, one has to buffer on producer

side as well and take precautions to ensure an atomic update of the consistent set e.g., by the use of semaphores. In Figure

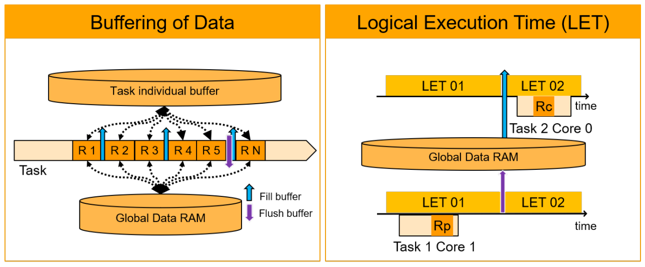


Figure 4. Left: Implementation of stability and coherency by usage of consistency buffers. Right: Implementation via the LET paradigm

4 the sequence of runnables in a task is shown on the left side. The arrows indicate buffer copy positions for filling and flushing of consistency buffers. The dotted lines symbolize data accesses. Depending on the need for stability or coherency the data is accessed either in the buffer or in the global shared memory. [4].

Another way to ensure stability and coherency is the execution of runnables in a controlled way. In this case, interference in the data access is prohibited by scheduling and synchronization points where needed, see Figure 4 right side. The paradigm of Logical Execution Time (LET) [6] is one example for this mechanism.

### C. *Impact on performance and memory size*

In a mid-sized engine management project (3MB code and parameter, 250kByte data) today one can find around 400 consistency requirements. Two thirds of them are addressing coherency of scalar data, the rest data stability. Ensuring data consistency requires various resources of the micro controller, such as RAM memory for consistency buffers, program memory and run-time for the copy routines. Furthermore, blocking times are caused by semaphores or synchronization points.

In the mentioned example project almost percent of the data project distributed on three cores. The implemented consistency buffers use around 11 KBytes of RAM and the fill and flush routines need around 7 percent of CPU-load to route almost 7500 accesses to buffered data instead to the globally shared data. These correspond to 12 per cent of the total amount of data accesses statically counted.

These figures consider already optimizations, which could be performed due to the knowledge of the consistency requirements of the modules. Without this knowledge, the amount of resources would have to be more than duplicated with a significant performance drawback. Therefore the specification of consistency requirements is necessary to allow this optimization.

### D. *Specification of Consistency Requirements*

With this situation in mind, several exchange formats for automotive embedded software foresee the description of data consistency requirements.

In AUTOSAR-Classic V4.2.2 [1] onward one can specify for a software component ConsistencyNeeds for stability of data for a RunableEntityGroup(s) and or coherency for DataPrototypeGroup(s). It is also possible to use the inverted logic and specify for which RunnableEntyGroup(s) stability of data is not required or for what DataPrototypeGroup(s) coherency is not needed.

ASAM MDX 1.3.0 and higher provides tags in the SW collection and in AMALTHEA in APP4MC 0.7.2 and higher uses DataCoherencyGroup to define these requirements [7]. In context of the engine management project mentioned in 1 consistency specification is done in almost all cases on consumer side. The consumer module specifies those requirements for stability or coherency towards its input data. In seldom cases the specification happens on producer side.

Besides the pure technical possibility to describe the requirement, the elicitation of the requirement is more important, especially in a domain, like power-train, where a large portion of the control software is developed by mechanical engineers.

## III. How to identify Consistency Requirements

### A. *Today's approach*

In our current process, identification of consistency requirements happens when a software module is designed, via a so-called design object review, which can be considered as a module test. The quality of requirements, their completeness and necessity, depend on the awareness and experience of the function developer and reviewer with respect to the problem of data consistency. As most of the function developers in power-train have a background in mechanical engineering, they tend to over-specify in order to be on the ”safe” side. Over-specification of such requirements could lead to:

- Rejection of requirements:
  The Multi-Core-Layer mentioned in figure 1 can only provide coherency if data is produced in the same task. If one would require coherency for all input variables of a runnable entity, this will most likely be rejected, as such variables are usually produced in different execution contexts and therefore cannot be provided in a coherent way.
- Functional misbehavior:
  Stability could lead to unexpectedly long response times. One could think of a handshake-flag with the intention to trigger as fast as possible an activity of a runnable in another context. If this flag would be buffered for stability the changed value would reach the consumer with one calculation cycle delay.
- Increased resource consumption:
  Every established consistency buffer needs RAM, ROM and Runtime for the buffer management.

To overcome these weaknesses we were searching for a formal method to elicit consistency requirements and complement the module test by that approach. This would support the platform aspect and result in a systematically tested module with respect to data consistency, the necessary requirements, and an increase in quality.

## B. Adversarial Consistency Testing

The execution of a component is triggered by some event (trigger) and takes place within a time interval bounded by a given worst-case execution time (WCET). Components are executed concurrently as if each of them were mapped to a dedicated processing unit (no preemption). In every execution, a component reads data from input variables and writes data to output variables. A variable represents a data container on which atomic read and write operations are performed. The scope of consistency requirements is one component, called *Module Under Test* (MUT) and a group of its input variables $C$, called a *consistency set*. We confine our attention to the case where all data in $C$ is produced by one component, called *Provider* (PRV). A run of the application is represented by two concurrent sequences of MUT and PRV executions.

The stability and coherency conditions considered in this paper refer to the time instant in a MUT execution when the first read to a $C$ variable is performed. Let $\widetilde{C}$ denote the snapshot of $C$ at such an instant. We are interested in:

- Stability: All values read from variables in $C$ must be drawn from $\widetilde{C}$.
- Coherency: All values in $\widetilde{C}$ are produced in the *same execution* of the provider, i.e., $\widetilde{C}$ is also the snapshot of $C$ at the termination instant of a provider execution.

We propose a test-based approach, where MUT code fragments are executed interleaved with PRV code fragments, in order to force occurrences of consistency violations, and also to propagate their effect on MUT outputs. If an input variable is involved in no violation, or if all violations involving the variable have no significant effect on the outputs, then the variable need not be buffered.

This is a type of adversarial scheduling that seeks to maximize occurrences of violations by manipulating execution times of code fragments to achieve "bad" interleaving of MUT and PRV executions. To this end, we employ the Validator simulator [8], where execution of application software is interleaved with simulation of a virtual platform model. The control flow jumps from application to virtual platform via instrumentation inserted just before variable accesses in the source code, at places called *access points*. At an access point, execution control is transferred to the platform simulation, together with the execution time budget associated to the fragment of code executed from the previous access point. This time value is calculated during execution by dedicated instrumentation. The platform simulates, among other things, the consumption of the time budget on a virtual CPU core. Thereafter, the execution switches back to the application (where the variable is accessed in the next step), and proceeds until the next access point is reached. In the case of consistency testing, the application comprises one MUT task and one PRV task per consistency set, the access points refer to MUT input variables, and the virtual platform model contains a two-core CPU: one for the MUT task and one for the PRV task(s).

In a standard SIL simulation, henceforth called a *nominal* run, a software component is executed as a unit in zero time at the moment when it is triggered. In a Validator-controlled run, an execution starts at the trigger time and may take a non-zero amount of (simulated) time, up to the specified WCET value. There are two manipulation policies, one for stability violations and the other targeted to coherency violations. The protocols are formally specified as timed automata that are composed with the existing Validator models of platform and task execution [9], and implemented as state machines which are executed in the access point instrumentation. The executable models are presented in the next section. Since the formal models cannot be detailed here for space reasons, we describe them informally below and then illustrate their workings by means of an example.

In the stability run, the MUT always performs the first access to a variable of a consistency set. Thereafter, when execution reaches another variable whose value has stayed unchanged since the first access, the MUT posts a message representing a request to modify the variable and waits for an answer. The PRV tries to satisfy such a request by executing until either it writes to the variable, or it terminates. If (when) the write happens, the PRV sends a positive answer to the MUT and goes (back) into the waiting state. Consequently, the MUT resumes execution. If the PRV write does not happen, the PRV terminates not before deciding whether to send a negative answer to the MUT or to leave the request to be handled by the next PRV execution. Waiting is always limited in time by the WCET deadline: if a component is waiting when its WCET expires, it wakes up and executes (instantaneously) until termination. The coherency run is similar, with the main difference that the MUT waits *before* the first access to the consistency set.

*Example:*
Consider a MUT with 5 input variables $a, b, c, d, e$, and two outputs $o_1, o_2$. All input values are produced by the same provider module PRV. The inputs are grouped in two consistency sets: $C_0 = \{a, c, e\}$ and $C_1 = \{b, d\}$. Both MUT and PRV are triggered periodically, with MUT's period = 4ms and WCET = 2ms, and PRV's period = 3ms, offset = 0.5ms, and WCET = 1ms. We confine our attention to the first hyperperiod, consisting of 3 MUT executions and 4 PRV executions.

In a nominal run, the (read) access sequences in the three MUT executions are $(c, a, b)$ at simulation time 0.0, $(d, b, e, c)$ at time 4.0, and $(e, c, d, b)$ at time 8.0. Also, the write sequences in the four PRV executions are $(a, d, c)$ at time 0.5, $(e, b)$ at times 3.5 and 6.5, and $(c, e, d, b)$ at time 9.5. The various test runs are illustrated in Figures 5 and 6 and explained in the sequel. In these figures, empty rectangles represent WCETs, filled rectangles represent (non-zero) execution times, and thick bars represent zero-time executions. Variable accesses are shown in each execution as untimed sequences, spread horizontally such that one can see the interleaving of reads in the MUT with writes in the PRV. Note that scaling of the time axis is non-uniform (to accommodate the various interleavings). The nominal run is depicted in Figure 5($i$). The first stability run is illustrated in Figure 5($ii$) and described next in timestamp order.

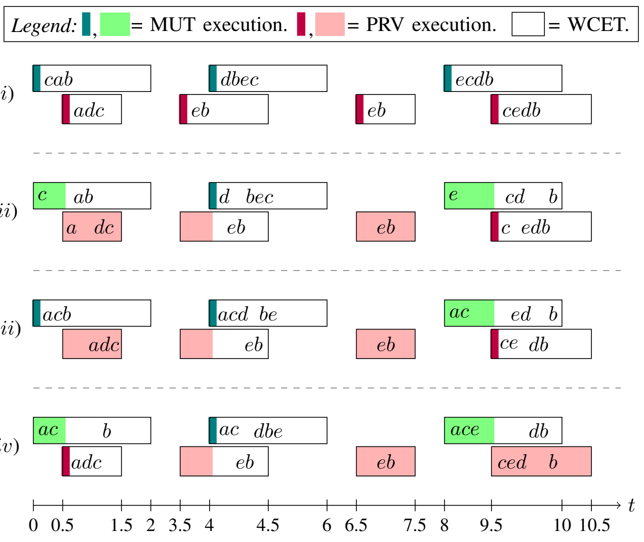


Figure 5. Testing runs with read/write accesses: nominal ($i$), stability ($ii$: unbuffered , $iii$: $a, c$ buffered) and coherency ($iv$: $a, c$ buffered)

0.0: MUT reads $c$, then waits before reading $a$.
0.5: PRV writes to $a$ and waits (before writing $d$). MUT proceeds, reads $a$ and $b$ and terminates (at simulation time 0.5). A stability violation involving variables $(c, a)$ from $C_0$ is reported.
1.5: As the PRV's WCET expires, PRV resumes and executes until termination (writing $d$ and $c$).
3.5: PRV is triggered and waits before accessing $e$.
4.0: MUT reads $d$, then waits. PRV wakes up and writes to $e$, then to $b$ (which satisfies the MUT's request), and terminates. MUT resumes, reading $b, e, c$, and terminates. Stability violation: $(d, b)$.
6.5: PRV waits until its WCET expires (at 7.5), when it executes completely.
8.0: MUT reads $e$, then waits before $c$.
9.5: PRV writes $c$ then waits before $e$. MUT resumes and reads $c, d$, then waits for $b$ to be changed. PRV wakes up and updates $e, d, b$, sends a positive answer and terminates. Stability violations: $(d, b)$ and $(e, c)$.

Note that the variables whose modifications caused the loss of stability are $a, c$ from $C_0$, and $b$ from $C_1$. Assume that a decision is made to buffer $a$ and $c$ in the MUT, while the case of $b$ is not yet decided. Thus, the user marks $a$ and $c$ as "protected" in the MUT. Consequently, the tool automatically modifies the software by including an atomic section to be executed at the trigger time of the MUT in order to fill the buffers, and redirecting the read accesses in the MUT to use the buffers. The buffer-fill section is not subject to execution switching: no waiting is done before or inside this section.

In the stability test with the new settings, shown in Figure 5($iii$), every MUT execution always starts with the atomic buffer-fill section for variables $a$ and $c$. The MUT execution at time 0.0 goes through without any waiting. In the executions triggered at 0.5 and 6.5, the PRV waits until its WCET expires, without achieving any violation. As for the other executions:
3.5: PRV waits before accessing $e$.
4.0: MUT fills $a$ and $c$ buffers, reads $d$, then waits before $b$. PRV wakes up and writes to $e, b$, then sends a positive answer to the MUT and terminates. MUT resumes execution, reads $b, e$, and terminates. Stability violations : $(d, b)$, $(c, e)$, $(a, e)$.
8.0: MUT reads $a, c$, then waits before $e$.
9.5: PRV updates $c, e$, then waits. MUT resumes and reads $e, d$, then waits before $b$. PRV updates $d, b$, and terminates. MUT reads $b$ and terminates. Stability violations: $(d, b)$, $(c, e)$ and $(a, e)$.

The consistency testing tool complements the list of achieved violations with output analysis results, obtained by comparing MUT outputs with reference signals from the nominal run. This enables the user to perform impact analysis and/or root cause analysis, and to factor the results in the decision making process. To illustrate this part in our example, we consider next that certain code fragments are executed as shown in Listings 1 to 4. For a run of the application, let $M^1, M^2, M^3, \ldots$ denote the sequence of MUT executions, and $P^1, P^2, P^3, \ldots$ denote the sequence of PRV executions. We assume that outputs $o_1, o_2$ are updated in executions $M^2$ and $M^3$ as shown in Listing 1, and Listing 2. The code fragments in Listings 1 and 2 sit on alternative execution paths of the MUT, one being exercised in $M^2$ and the other one in $M^3$. For PRV executions, we limit our attention to how variables $e$ and $b$ are set in executions $P^2$, $P^3$ - as shown by Listing 3, and in execution $P^4$ - see Listing 4.

Listing 1. Original MUT fragment executed in $M^2$

```
tmp1 = d;
 if (b > 0) {
   o1 = tmp1;
   }
tmp2 = e;
o2 = c;
```

Listing 2. Original MUT fragment executed in $M^3$

```
 if (e > 0) {
   o2 = c;
   }
o1 = d;
tmp3 = b;
```

Listing 3. PRV fragment executed in $P^2$ and in $P^3$

```
e = 1;
b = 1;
```

Listing 4. PRV fragment executed in $P^4$

```
e = 0;
b = 0;
```

For these execution paths, one can determine that the output values in the previous stability run (Figure 5($iii$), with $a$ and $c$ buffered) are the same as in the nominal run: Note first that the violations involving $a$ have no effect as the value read from $a$ is never used in $M^2$ and $M^3$. Also, the values of $b, c, d, e$ read in $M^2$ in the stability test are the same as the one read in the nominal run (i.e., those produced by $P^1$ and $P^2$). In contrast, the value read from $e$ in $M^3$ is different than the one in the nominal run, (it is written by $P^4$ instead of $P^3$). According to Listings 2 and 4, this has the effect that $o_2$ is not updated, so its value is the same as the one set in $M^2$. In the nominal run, $M^3$ updates the output with the very same value. Thus, violation $(d, b)$ has no effect at all, while violation $(c, e)$ changes the execution path but not the output value.

In conclusion, it is sufficient to buffer only variables $a$ and $c$ to achieve relevant data stability. If coherency of input values is ensured at the start of every MUT execution, then consistency is obtained (with the reduced buffering).

Consider next the coherency requirement. Note that some stability violations are also coherency violations - for example, $(c, e)$ from the previous run has $c$ produced in $P^1$ and $e$ in $P^4$. To enforce further coherency violations, a dedicated protocol is employed: The MUT execution is paused before the first access to a consistency set, when the message to modify the involved variable is sent. The corresponding run (with variables $a$ and $c$ buffered in the MUT) is depicted in Figure 5$(iv)$. We describe next only the last MUT and PRV executions, as they are the only ones that result in new (coherency) violations.

8.0: MUT reads $a, c, e$, then waits before $d$.

9.5: PRV updates $c, e, d$, then waits. MUT resumes, reads $d, b$, and terminates. Coherency violation: $(d, b)$.

10.5: End of WCET for the PRV, which writes $b$ and terminates.

Notice that coherency violation $(d, b)$ in run $(iv)$ is quite different than stability violation $(d, b)$ in run $(iii)$. While the latter has no effect on MUT outputs, the former affects output $o_1$, which is updated in run $(iv)$ with the value of $d$ produced in $P^4$ rather than the one provided by $P^1$, as in the nominal run. Consequently, variable $d$ is configured as "protected" in the MUT. The tests are now repeated with $a, c$, and $d$ buffered in the MUT, with the two runs depicted in Figure 6.

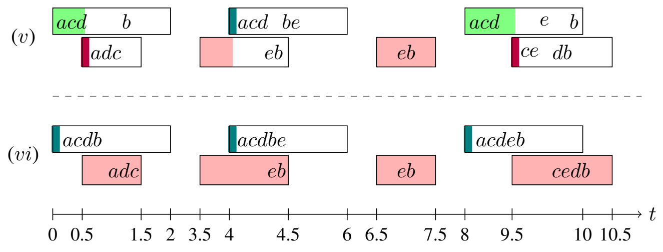


Figure 6. Further runs for stability $(v)$ and coherency $(vi)$ with $a, c$, and $d$ buffered

In the stability test (Figure 6$(v)$) the first MUT execution waits before accessing $b$ until time 0.5, when the PRV executes completely without creating any violation. In $M^2$, the MUT waits before $b$, and we get violations $(d, b)$ and $(c, e)$. The same violations are incurred in $M^3$. As in run $(iii)$, these stability violations change nothing in the MUT outputs as compared to the nominal run, hence no new buffering requirement is generated. In the coherency run (Figure 6$(vi)$), no waiting is done in the MUT, as $a$, $c$, and $d$ are read at the trigger time and the accesses to $b$ and $e$ are not first accesses to consistency sets. Hence, no further coherency violation is obtained.

In conclusion, variables $b$ and $e$ do not need buffering. Note that the ability to (automatically) compare outputs with the standard SIL simulation is crucial for the practical application of this approach. The result of output checking can form the basis of an exception (no buffering despite violations), and/or it can guide a root cause analysis on the software.

We proceed next to formally describe the stability and coherency requirements considered in this paper. Let $t_s$ denote the start time of an execution and $t_e$ the termination time. The following assumptions are made to simplify the presentation and without loss of generality: MUT executions are sequential: $t_s(M^i) > t_e(M^{i-1})$, exactly one PRV updates a MUT input variable, PRV executions are sequential: $t_s(P^j) > t_e(P^{j-1})$, and all input variables are read in a MUT execution.

Let $C$ be a consistency set (group of input variables): $C = (q_1, \ldots, q_n)$. We use $q(t)$ to denote the value of variable $q$ at time $t$. We refer to the sequence of read accesses to variables in $C$ in execution $M^i$ as $Read^i(C) = (r_1^i, \ldots, r_m^i)$ where $r_k$ is the $k-th$ read performed in program order: $r_k = (q_k, t_k)$ represents the accessed variable $q_k \in C$ and the time of access.

The stability property requires that the values read in the MUT from the variables in $C$ are drawn from the snapshot of $C$ at the time of the first access to $C$:

*Definition 1:* The *stability* property is satisfied in MUT execution $M^i$ if the read sequence $Read^i(C)$ is such that

$$q_k^i(t_k^i) = q_k^i(t_1^i), k = 2, m$$

Under this stability requirement, a sufficient condition for consistency is that the data in $C$ is coherent at the time instants $t_1^i$, $i = 1, 2 \ldots$, with coherency criteria specified separately.

For the case when all values in $C$ are produced by the same component PRV, we consider the following coherency condition. Let us first denote by $Write^j(C) = (w_{q_1}^j, \ldots, w_{q_n}^j)$ the list of values written to the variables in $C$ during the execution $P^j$. Even if a variable is not actually accessed in writing, its (constant) value during $P^j$ is still listed in $Write^j(C)$.

*Definition 2:* The *coherency* property is satisfied in MUT execution $M^i$ if the read sequence $Read^i(C)$ is such that

$$\exists P^j \text{ with } t_s(P^j) < t_e(M^i) \text{ and } q_k^i(t_k^i) = w_{q_k}^j, k = 1, m$$

A *stability violation* in execution $M^i$ is a pair $(r_1^i, r_k^i)$ with $q_k^i(t_k^i) \neq q_k^i(t_1^i)$, representing the fact that the value of variable $q_k^i$ was changed between the time $t_1^i$ of the first read access to $C$ and the time $t_k^i$ when the variable $q_k^i$ is read in execution $M^i$. Note that if the violation involves the same variable ($q_k^i = q_1^i$) then this is clearly a consistency violation, since stability of atomic data is compromised. A stability violation involving distinct variables may or may not be also a consistency violation, depending on coherency considerations. However, the decision to issue a buffering requirement can be made also in the lack of coherency information, by comparing the simulation outputs with those from the nominal run. If significant differences are observed and traced back to the stability violation, then this is sufficient motive to request buffering of the involved variables. If a violation does not have a significant effect on the outputs, then no buffering request is made, regardless of coherency.

A *coherency violation* occurs in execution $M^i$ if values of at least two members of $Read^i(C)$ are produced in different provider executions: $\exists k, l \in 1, ..., m$ s.t.

$$q^i_k(t^i_k) = w^h_{q_k} \text{ and } q^i_l(t^i_l) = w^g_{q_l} \text{ for some } h, g \in \mathcal{N}, h \neq g$$

## *C. Modeling and design of the scheduling protocol*

The scheduling protocol is defined by means of MUT and PRV timed automata according to the standard task execution model, where a task can be in one of the following states: suspended, in execution, waiting, or preempted. This model is also employed in the Validator simulator [9], which is the target execution environment for the scheduling protocol. Since the Validator was implemented in C, this was also the target implementation language for the protocol. The end result was a tool developed in a model-based design process briefly presented below. We include also description of executable models with the aim to complete here the presentation of the scheduling protocol.

The initial specification expressed as timed automata was first translated into actors in the Modal Model domain in Ptolemy II modeling and simulation environment [10]. This enabled quick prototyping and validation by simulation. Thus, the actors are run within a top-level Ptolemy II test model with test cases (manually created) that provide inputs consisting of MUT and PRV executions specified as access sequences, trigger times and WCETs (no actual functionality is included). The detected violations are then checked against expected ones.

To verify the Validator-integrated C implementation, a new actor was added to the Ptolemy test model, enabling co-simulation with the Validator. The model is shown in Figure 7. The VALIDATOR_COMM actor ensures that the Validator simulation (executed as a separate process/thread) is done in lockstep with the MUT and PRV controller actors in Ptolemy, on the same test inputs. The CT_BACKEND actor implements conditions and actions invoked from the transitions of the MUT and PRV modal models, as well as verification of violations reported by the Validator run against the ones detected by the MUT_CONTROLLER actor. By including randomization in the actors that generate the test inputs for MUT (MUT_TRIGG and MUT_accSeq) and for the PRV (PRV_TRIGG and PRV_accSeq), the Ptolemy model essentially generates test cases and verifies them online, recording only the failed ones.

The top-level state machine of the MUT controller actor for stability testing is shown in Figure 8. The modal models described in the sequel have been simplified for the purpose of presentation clarity. The actual transition conditions (guards) and actions (in Ptolemy II syntax) are hidden, only their descriptive text annotations being shown on the transitions. The guards that enable/disable transitions are written between square brackets (a missing guard means the transition is always enabled). Due to space concerns we do not include here the

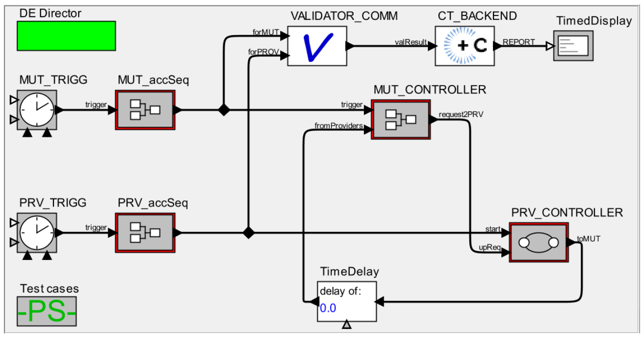


Figure 7. Implementation testing model in Ptolemy II

meaning of graphical annotations on the transitions, which can be found in [11].

The initial state is suspended (S). A MUT execution is triggered when the input port $trg$ contains a Ptolemy data token; then the transition from state S to state E (execution) is taken and protected MUT variables are buffered. State E contains a refinement actor that is initialized with the access sequence specified in the token removed from $trg$. The E refinement actor simply visits the sequence elements in turn until the end of the sequence, when the ”Execution terminated” guard enables the transition to the suspended state. A sequence element specifies the access type (read/write) and the variable involved. When an element is visited, one of the three access transitions outgoing from E is enabled. The most interesting one is the transition to state W (waiting), representing the fact that the next execution step is a read access to a MUT input variable where waiting may be needed. The refinement of state W is shown in Figure 9.

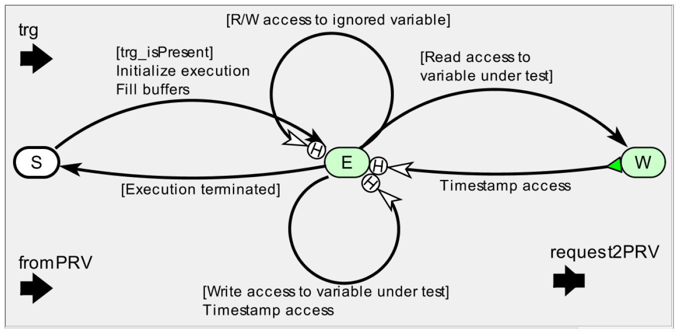


Figure 8. MUT state machine (top level)

From the initial state, a check is made whether this is a first access to a consistency set or not.

- If yes, then the transition to the final state is taken, which in turn enables the top-level transition from W to E, which records the access and its timestamp. Then E continues with the next element of the access sequence.

- Otherwise (i.e., second access), the local state chages to ”check4PRV” and a look-ahead is performed to determine whether a PRV is/will be executed until the current WCET

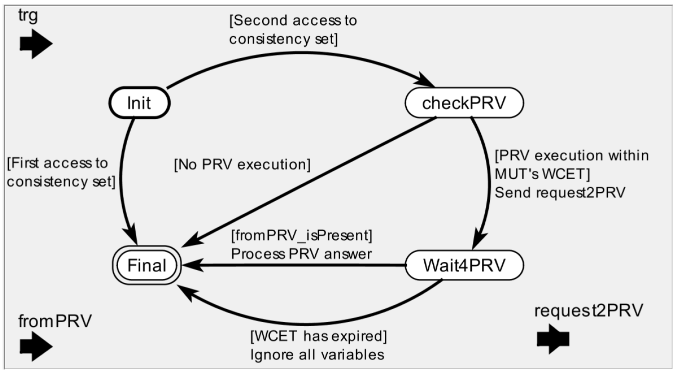


Figure 9. Refinement of state W: processing of read access

window of the MUT expires. If so, then the transition to state ”Wait4PRV” is taken and a token (representing a modification message) is sent on the ”request2PRV” output port (connected to the PRV controller as seen in Figure7). If (when) an answer from the PRV is received - as a token on the input port ”from PRV” - the answer is processed and if it is positive, then a violation is recorded. Note that most of the conditions and actions on the transitions are implemented as methods of the top-level actor CT_BACKEND, responsible also with bookkeeping and reporting. If no answer comes before the WCET expires, the lowermost transition is taken and all the remaining accesses in the current execution are marked to be ignored.

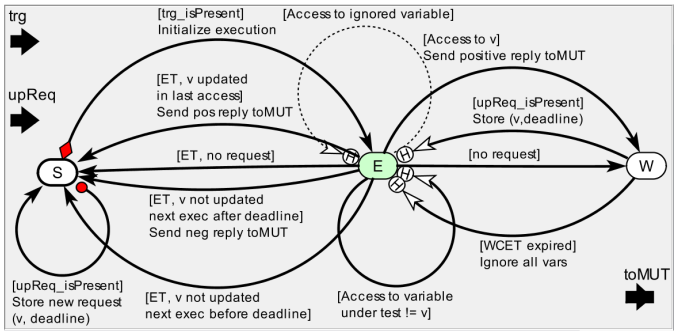


Figure 10. Provider model

The PRV controller model is shown in Figure 10. A MUT message comes as a token on the input port $upReq$, containing the involved variable $v$ and a deadline for the request, which is the expiration time of the MUT's WCET. A token received in the suspended state is simply stored to be processed in the next execution. The refinement of state $E$ is the same as for the $MUT$, where a sequence element represents a write access. If no request is present when PRV is triggered, then the PRV goes into the state $W$ (waiting for request). Otherwise, if the accessed variable is different than the requested one ($v$), then the access is timestamped. Upon writing to $v$, a positive answer is sent (as a token on the $toMUT$ port) and the state switches to $W$. The PRV goes from the waiting state into the execution state when a new request is received, or when its WCET expires. In the latter case, all further accesses are ignored (only the dotted transition in state E will be enabled). The four transitions from $E$ to $S$ represent alternative conditions and corresponding actions at the execution termination of the PRV (denoted by ET in the figure). From top to bottom: A positive reply is sent to the MUT if the last access before termination satisfies the request. Nothing is done if no request is present when termination occurs. If a request is pending when the PRV terminates, and if the time of the next PRV trigger falls after the request's deadline, then the request cannot be satisfied (negative answer to MUT). Otherwise, the request is left pending in the hope that it may be satisfied in the next execution.

## *D. Application and consistency testing workflow*

We consider a Simulink SIL model where the MUT and possibly PRV software are included as an S-function, to which we associate a specification of consistency sets and (worst-case) execution times. The model may contain only the MUT, in which case a generic PRV code is generated by the tool chain, such that a PRV task samples the system inputs in an order specified in the configuration. Based on the initial consistency specification, a Validator build tool configures the virtual platform, generates PRV source code if needed, then instruments all the source code at the access spots and builds an executable (see Figure 11). Furthermore, the tool also generates an S-function that replaces the (standard) SIL S-function in the Simulink model. Upon starting a simulation, the Validator executable is launched as a separate process, performing co-simulation in lockstep with the Simulink model, recording all the inputs and the outputs on the file system. After the co-simulation ends, the Validator program may continue to perform batch simulations for combinations of alternative consistency sets and execution times, using always the same input values, as recorded from the initial co-simulation. Two such runs are performed: one for stability violations and the other for coherency violations. All outputs are checked against reference signals from the standard SIL simulation, and all results are aggregated and presented to the user at the end of the two runs.

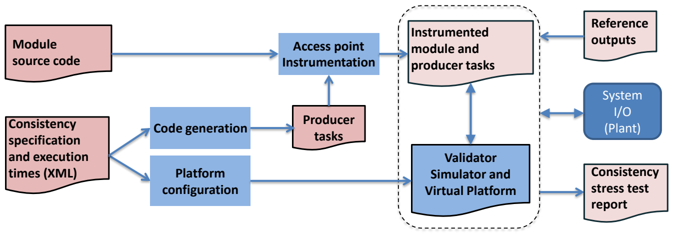
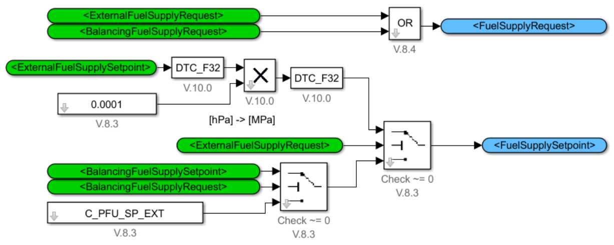


Figure 11. Testing workflow

## IV. Case Study

### A. Example control function

In this section we describe the application of the test on a typical module of a power-train control function, which can be seen in Figure 12. The picture shows one part of a bigger functionality. The function selects, based on the two discrete inputs variables ExternalFuelSupplyRequest, in short *ExtReq*, and BalancingFuelSupplyRequest, in short *BalReq*, either the contiguous input variables ExternalFuelSupplySetpoint, in short *ExtSetp*, or BalancingFuelSupplySetpoint, in short *BalSetp*, as the output variable FuelSupplySetpoint.



<ExternalFuelSupplyRequest>
<BalancingFuelSupplyRequest>
OR
V.8.4
<FuelSupplyRequest>
<ExternalFuelSupplySetpoint>
DTC_F32
V.10.0
V.10.0
DTC_F32
V.10.0
0.0001
V.8.3
[hPa] -> [MPa]
<ExternalFuelSupplyRequest>
<FuelSupplySetpoint>
Check ~= 0
V.8.3
<BalancingFuelSupplySetpoint>
<BalancingFuelSupplyRequest>
C_PFU_SP_EXT
V.8.3
Check ~= 0
V.8.3

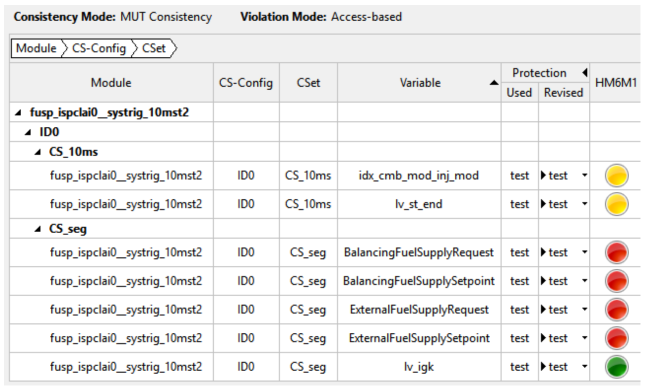
Figure 12. Typical example of a power-train control function

This function is called time periodic, every 10ms. The input values are produced in the so-called segment task for every combustion cycle of the four stroke engine. Thus, the period of this task will change according to the engine speed and number of cylinders. For a four cylinder engine at 6000rpm the calculation period will be 5ms, while at 4000rpm it will be 7.5ms. For the consistency testing we have to assume that this task might be executed concurrently on another core than the 10ms function of the module under test.
The expected behaviour of the output is a coherent switch between *ExtSetp* and *BalSetp* values, according to the setting of the request flags *ExtReq* and *BalReq*, with the former having priority on the latter when both flags are set and a default value being used when no flag is set.

### B. Test for coherency of scalar variables

In the absence of a software provider for the MUT inputs, the tool automatically generates provider tasks that simply sample the input signals from the Simulink model. The user can configure a PRV task to be triggered by a plant signal, by a random pulse with settable minimum and maximum widths, or by a change in the inputs.

After running the stability and coherency tests, the user is presented with a summary report of violations as well as with output comparison plots. An example of consumer-stability results for the entire module is given in Figure 13 (for one execution time scenario), where one can see that the inputs shown in Figure 12 are all involved in violations, while another variable in the same consistency set is not. Two other variables (different consistency set) are marked yellow, meaning they have not been accessed during simulation.



**Consistency Mode:** MUT Consistency **Violation Mode:** Access-based

Module > CS-Config > CSet

| Module | CS-Config | CSet | Variable | Protection | | HM6M1 |
|---|---|---|---|---|---|---|
| | | | | Used | Revised | |
| **fusp_ispclai0__systrig_10mst2** | | | | | | |
| **ID0** | | | | | | |
| **CS_10ms** | | | | | | |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_10ms | idx_cmb_mod_inj_mod | test | test | 🟡 |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_10ms | lv_st_end | test | test | 🟡 |
| **CS_seg** | | | | | | |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_seg | BalancingFuelSupplyRequest | test | test | 🔴 |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_seg | BalancingFuelSupplySetpoint | test | test | 🔴 |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_seg | ExternalFuelSupplyRequest | test | test | 🔴 |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_seg | ExternalFuelSupplySetpoint | test | test | 🔴 |
| fusp_ispclai0__systrig_10mst2 | ID0 | CS_seg | lv_igk | test | test | 🟢 |

Figure 13. Summary of the data consistency test for the example module

As a trigger signal for the segment task was not available in the plant model, the tool was configured to generate the PRV trigger at every change in the inputs *ExtReq* and *BalReq*. An output plot example for a coherency run is shown in Figure 14(top). The red signal is the nominal output and the blue one is the output of the manipulated run. The middle and bottom plots depict the external and balancing inputs, respectively. The nominal behavior is as follows: until time 2.84s, the output switches between the value of *ExtSetp* (when *ExtReq* is set) and a default value of 20 (when *ExtReq* is reset). After 2.84s, when *BalReq* is set, the output switches between *ExtSetp* (when *ExtReq* is set) and the value of *BalSetp* (when *ExtReq* is reset). Note that *ExtReq* has a rising edge at time 2.85s. In the manipulated run, every time one of the request signals is set, the output drops to zero for 10ms, with an accumulated effect between 2.84s - 2.86s. This illegal behavior is a result of forced coherency violations. For example, at time 2.6s, both MUT and PRV are triggered and: (1) the MUT waits for the PRV to modify *ExtReq*; (2) PRV sets *ExtReq* and then waits before setting the new value of *ExtSetp*; (3) MUT proceeds to read the old value of *ExtSetp* (zero) and, since *ExtReq* is set, it updates the output with that value; (4) PRV updates *ExtSetp* with the new value. The next MUT execution comes after 10ms and puts the new value into the output. Similarly, at time 2.84s the MUT will update its output with the previous value of *BalSetp* (zero) instead of the most recent one. This is repeated in the next execution at 2.85s due to the *ExtReq* input that has priority, thus yielding the 20ms interval of incorrect output.

### C. Effort considerations

The stress test in the SIL for the above use case was run on an INTEL i7 running at 2.7 GHz with 32GB RAM. The preparation of the SIL environment needs about one hour, one-time time effort. Executing one test case takes in the range of 12 seconds for a simple model up to 3.5 min time for a complex one. Most of the models we see in our application would need 1 minute test execution time. In average we see 5 test cases per module. Finally the evaluation of the test results needs less than 30 minutes. Typically a module is reused in the

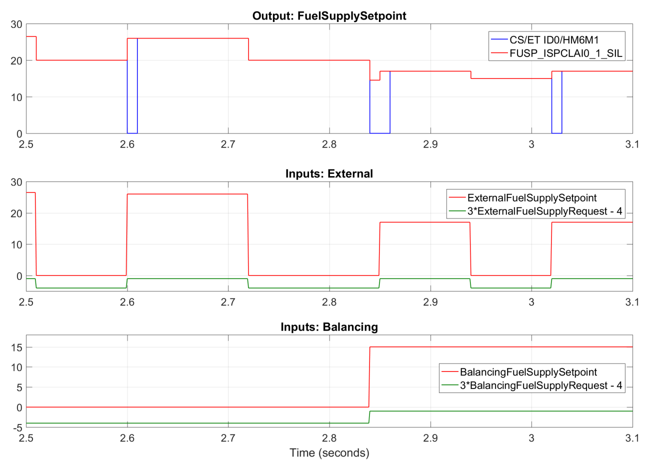


Figure 14. Output signal comparison (top), and corresponding inputs

same version in 10 projects. The total time for the consistency stress test for one module with a platform coverage under these conditions is expected to take 140 minutes (60 minutes plus 1 minute multiplied by 5 test cases multiplied by 10 projects and 30 minutes of evaluation of test reports.) In case of changes, the set-up time can be neglected, therefore mainly half an hour report evaluation is needed.

## V. Related Work

The general problem of data consistency in real-time systems has been addressed quite extensively at the system design level, with approaches ranging from real-time data bases to schedule synthesis. Buffer optimization is a main concern in scheduling of dataflow graphs, see e.g. [12], [13]. Here we have the same objective but in a different setting, as the function developer has no control over scheduling.

The approach presented in this paper can be regarded as a type of cooperative scheduling, where a task voluntarily relinquishes execution control at chosen steps in its execution. While cooperative scheduling has been employed to avoid failures in multithreaded programs (see, e.g., [14]), here it is used for the opposite purpose. We also mention the tool Symbiosis [15], which creates failing multithreaded schedules to (re)create bugs in a multithreaded program. Another example of related bug detection tool is AVIO [16].

Testing of concurrent programs is a related and extensively studied area. An comprehensive survey in this respect is provided in [17], which groups the existing results in several categories: reachability, structural, model-based, mutation-based, slicing-based, formal methods, random testing, and search-based testing. For example, the work in [18] deals with unit testing of multithreading code by enforcing schedules for multithreaded tests. An example of random testing approach that seeks to create race conditions with high probability is given in [19]. Our proposal is a combination of model-based, slicing-based, formal methods and search-based testing for elicitation of data consistency requirements in real-time multicore applications.

## VI. Summary and Outlook

This method has been applied with success on several modules of an engine management system. With this we are able to support the function developer with an automatized stress test for the identification of consistency requirements, which will improve the quality of those requirements, as today those are strongly depending on the knowledge of the function developer concerning data consistency. Vitesco Technologies intends to integrate this method seamlessly into the existing MIL, SIL and PIL test chain.

## References

[1] AUTOSAR-Consortium, in *http://www.autosar.org*, Rev 4.2.2.

[2] U. Margull, M. Niemetz, and G. Wirrer, "Quirks and challenges in the design and verification of efficient, high-load real-time software systems," in *Proceedings of the 5th Embedded Real Time Software and Systems Conference (ERTSS)*, 2010.

[3] D. Claraz, F. Grimal, T. Ledier, R. Mader, and G. Wirrer, "Introducing multi-core at automotive engine systems," in *ERTS2*, 2014.

[4] R. Mader, A. Graf, G. Winkler, and D. Claraz, "Autosar based multicore software implementation for powertrain applications," in *SAE Worldconference*, 2015.

[5] R. Sieber, "Applying autosar in a powertrain dynamic architecture using multicore ecus," in *Embedded Multi Core Conference (EMCC)*, 2016.

[6] S. Resmerita, A. Naderlinger, and S. Lukesch, "Efficient realization of logical execution times in legacy embedded software," in *MEMOCODE17 Vienna*, 2017.

[7] D. Claraz, R. Mader, T. Flämig, and L. Michel, "Shared sw development in multicore automotive context," *Proceedings- ERTS 2016*, vol. 1, 2016.

[8] S. Resmerita and W. Pree, "Verification of embedded control systems by simulation and program execution control," in *2012 American Control Conference (ACC)*, June 2012, pp. 3581–3586.

[9] S. Resmerita, P. Derler, and E. A. Lee, "Modeling and simulation of legacy embedded systems," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-38, Apr 2010.

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[11] E. A. Lee, "Finite state machines and modal models in ptolemy ii," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-151, Nov 2009.

[12] S. Kang, D. Kang, H. Yang, and S. Ha, "Real-time co-scheduling of multiple dataflow graphs on multi-processor systems," in *53nd Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[13] N. Pontisso, P. Quèinnec, and G. Padiou, "Analysis of distributed multi-periodic systems to achieve consistent data matching," in *10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, May 2010, pp. 81–88.

[14] B. Lucia and L. Ceze, "Cooperative empirical failure avoidance for multithreaded programs," in *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 39–50.

[15] N. Machado, D. Quinta, B. Lucia, and L. Rodrigues, "Concurrency debugging with differential schedule projections," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 2, pp. 14:1–14:37, Apr. 2016.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2006, pp. 37–48.

[17] V. Arora, R. Bhatia, and M. Singh, "A systematic review of approaches for testing concurrent programs," *Concurr. Comput. : Pract. Exper.*, vol. 28, no. 5, pp. 1572–1611, Apr. 2016.

[18] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011, pp. 223–233.

[19] K. Sen, "Race directed random testing of concurrent programs," in *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2008, pp. 11–21.