# Application Independent Modeling and Simulation Environment for Systems with Self-aware and Self-expressive Capabilities

Tatiana Djaba Nya, Stephan C. Stilkerich
Airbus Group Innovations
Airbus Group GmbH
Ottobrunn, Germany
Email: {tatiana.djabanya, stephan.stilkerich}@eads.net

***Abstract*—Self-awareness and Self-expression in computer systems promise a lot of abilities enabling us to deal with the problems and challenges caused by their continuously increasing complex and heterogeneous structures and requirements, and the unpredictability and changes in their deployment environment. For this reason, engineering self-awareness and self-expression in computing systems has become a major research field in the computer science. To fill the gap between research at the conceptional level and the construction of first proof-of-concept demonstrators, a novel modeling and simulation environment for self-aware and self-expressive systems has been implemented. The environment is the Transaction-Level-Modeling (TLM) description in SystemC of the reference architectural framework for self-aware and self-expression systems. Therefore, it enables to simulate any topology of self-aware and self-expressive systems and deployed applications. This paper presents the said environment along with the developed reference architectural framework on which it is based, as well as an example motivated in the avionic domain.**

***Keywords-SystemC; Transaction-Level-Modeling; Simulation; Self-awareness; Self-expression.***

## I. INTRODUCTION

Self-awareness and self-expression, which is adaptive behaviour based upon it, have proven to have a lot of benefits for computing systems [1]. A computing system with self-aware and self-expressive capabilities is for example able to deal with unpredictability and changes in its deployment environment; it is also possible to implement more functionality in such a system, such that it has the ability to execute the corresponding functions according to the knowledge it has of itself and its environment. Therefore, the engineering of self-awareness and self-expression in computing systems has become an emerging and major research field over the past years. In that regard, there are some very important issues or questions like: what are the requirements of a self-aware and self-expressive computing systems? How to properly engineer self-awareness and self-expression capabilities in a computing system? How to ensure the correctness of a system after self-adaptation operations? How to ensure and maintain the reliability, the fault tolerance level in a system after self-expression? [2][3][4].

To be able to address these questions and many others in this context, a reference architectural framework which structures the requirements of a self-aware and self-expressive system has been built. Based on this reference architecture, a modeling and simulation environment that can serve as support and test environment for the development and demonstration of developed concepts has been implemented. An alternative concept for realizing fault-tolerance in avionic systems using self-awareness and self-expression that has been developed has been used to validate the environment.

This paper presents the previous mentioned elements and is organized as follows: Section II describes the self-aware and self-expressive architectural framework. Section III presents the modeling and simulation environment. Section IV presents an example case of this environment representing a one single node avionic system and showing the alternative idea of fault tolerance for avionic systems. Finally, Sections V concludes the paper.

## II. THE REFERENCE ARCHITECTURAL FRAMEWORK

Inspired from the biology and cognitive science, we defined in the EPiCS project [2] working definitions for self-awareness and self-expression in the context of a computing node [5][6]. Underlying these working definitions, we then developed the proposed reference architectural framework for a self-aware and self-expressive computing node [7]. This framework is shown here in Figure 1 and represents the conceptual components of a computing node with self-ware and self-expressive capabilities. Here, conceptual means that these components don't need to physically exit as separate components within an application, but provide a logical structure for reasoning about interactions between parts of a system, where these parts can have different levels of knowledge, autonomy and distributed decision making.

### A. Self-Awareness

Self-awareness is achieved in this architecture by the sensors, the private and the public self-aware engines. As in agent architecture, the sensors are used here to collect information. Two types of sensors can be distinguished: first, the internal sensors named "Sensor" in the architecture, which collect information about the node internal state and second, the external sensors which collect information about the node's context. These are represented in the architecture by the conceptual components named "environment" and "Other nodes". Both, the private and public self-aware engines are responsible here for collection of information from the corresponding type of sensors and for the processing of this information. Moreover they will trigger the node's reaction and adaptation process after the information processing, if necessary. According to the working definition, a self-aware system may possess historical knowledge, predictors of future likely states or contextual

 ISBN: 978-1-61208-341-4 

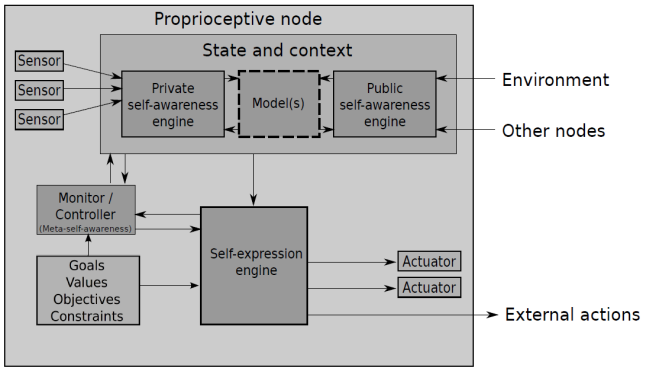


Figure 1. The conceptual components of a self-aware and self-expressive node.

information, in addition to purely instantaneous sensor readings. To enable this richer form of self-awareness, the self-awareness engines may engage in learning or modeling of information. We therefore introduce the possibility of internal models (online learning schemes), which are located in the component called ”Model(s)” and may be introduced as and when required in enabling the required level of self-awareness.

### B. *Self-Expression*

To achieve self-expression, our architecture includes actuators and a self-expressive engine. The self-expressive engine has the role of taking decisions about the actions that must be performed by the node itself in order to adapt its behaviour. As required by the working definition, this decision making process always take into account the node’s state, context, goals, values, objectives and constraints which are available in the node and later in this engine through the conceptual component bearing the same name. The actions determined by the self-expressive engine are passed to the actuators which execute them. Actuators are represented in this architecture by the conceptual components called ”Actuator” and the ones called ”External actions”. As its name suggests, the latter execute the actions targeting the node’s environment. The ”actuators” for their part executed the chosen actions targeting the node itself.

### C. *Meta-Self-Awareness*

Meta-Self-awareness is the higher level of self-awareness in a computing node and represents the ability of the node to be aware of its own awareness and to choose the level of awareness suitable to the node situation, to better achieve its goals. For this purpose, there is on the one hand a conceptual component named ”Monitor/Controller” in the reference architecture of the node. As shown in Figure 2 through the arrows, this component has access to the node’s goals, values, objectives and constraints, to the self-aware and self-expressive engines. In this way, it has a high-level view over the node’s behaviour and can intervene, when necessary, to lessen or increase the level of self-awareness and self-expression in the node.

## III. The Modelling and Simulation Environment

The modelling and simulation environment has been implemented in SystemC at the transactional level, i.e., Transaction-Level Modelling (TLM). TLM is a modelling methodology which is primary concerned with the efficient modelling of bus systems and their transactions (hence the name TLM). It reaches a higher abstraction level over the register transfer level

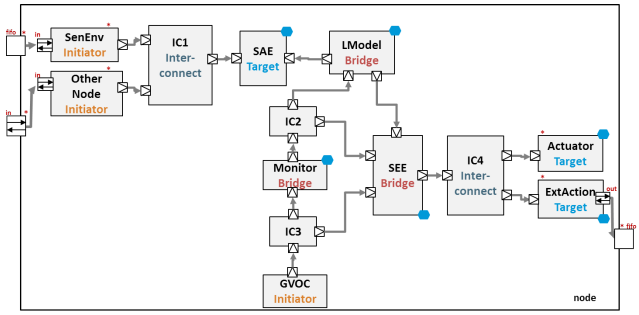


Figure 2. SystemC-TLM graphical view of the environment.

modelling and thus enables to implement virtual prototypes of systems which can be used to test developed software or to assess the performance of different system architectures through simulation. SystemC is a system description language. It enables both software and hardware description. The above mentioned properties and advantages of SystemC and TLM are the reasons which lead our decision to choose them for the realization of our modeling and simulation environment. The implemented SystemC TLM model is described in the following subsections. The next subsection gives an brief introduction in SystemC TLM in order to facilitate the understanding of the subsequent subsections focussed on the detailed proper description of the environment.

### A. *Theoretical background*

A SystemC TLM Model is mainly composed of components which communicate among each other over sockets by initiating transactions by the means of processes.

A component has a role which can be of three types: initiator, target and interconnect. An initiator is able to initiate transactions to communication with other components; a target cannot initiate transactions and is always the target of a transaction. As for the interconnect component, it functions as a bus or a router for the transactions and usually execute address mapping operations. A component can act as an initiator for some transactions and as a target for others. In this case it is called a bridge.

As mentioned above, the communication in a SystemC TLM model occurs here in form of transactions (method calls) through which, in its simplest form, an initiator has the possibility to write or read data to/from its target component. The details of the transactions such as the size, the address and the type of data are regulated by the initiator when initiating the transactions and later by the interconnect components, if present, to ensure the correct routing of data.

To be able to initiate transactions, initiators need thread processes. A process describes the functional behaviour of a TLM component. The SystemC simulator implements a cooperative multitasking environment, i.e. some process instances execute without interruption, only a single process instance can be running at any time, and no other process instance can execute until the currently executing process instance has yielded control to the kernel. A process shall not pre-empt or interrupt the execution of another process. Each process has a sensitivity list which is a set of events and time-outs which can be defined during implementation to determine when it is executed or resumed by the scheduler.

### B. *Model description*

The objective of achieving the functionality described in the reference architecture (Section II) of a self-aware and self-

 ISBN: 978-1-61208-341-4 

TABLE I: DESCRIPTION OF THE TLM MODEL

| Component | Role | Equivalent in the architecture | Transactions target(s) | Process(es) | Execution order | Transactions type |
|---|---|---|---|---|---|---|
| SenEnv | Initiator | Sensor, Environment | SAE | B | 3 | WRITE |
| OtherNode | Initiator | OtherNode | SAE | B | 4 | WRITE |
| GVOC | Initiator | Goals - Values - Objectives - Constraints | Monitor<br>SEE | A1<br>A2 | 1<br>2 | WRITE |
| LModel | Bridge | Model(s) | SAE<br>SEE | C1<br>C2 | 5<br>7 | READ<br>WRITE |
| SEE | Bridge | Self-expressive Engine | Actuator<br>ExtAction | D | 9 | WRITE |
| Monitor | Bridge | Monitor/Controller | LModel<br>SEE | E1<br>E2 | 6<br>8 | READ |
| SAE | Target | Private and Public Self-aware Engines | | | | |
| Actuator | Target | Actuator | | | | |
| Extaction | Target | External actions | | | | |
| IC1, IC2, IC3, IC4 | Interconnect | | | | | |
| node | Module | | | F | | |

expressive node as well as the data flow among its components in compliance with the rules and mechanisms of TLM lead us to the model presented in Figure 2. Complementary to Figure 2, Table I clearly listed the TLM components of this model, their roles, their processes, the transaction types and most importantly their equivalent in the reference architecture and the execution chronology of processes or rather transactions among them.

*1) Functional Behaviour of components:*
The first component to come into operation inside a node when the simulation is started is the Goals-Values-Objectives-Constraints (GVOC) component. As shown in Table I, it embodies the node's goals, values, objectives and constraints. It is an initiator and as such, it forwards the goals, values, objectives data respectively to the monitor through process A1 and to the component SEE through process A2. For this purpose, each of both processes A1 and A2 initiates WRITE transactions to the corresponding targets. The interconnect components IC3 placed between the GVOC component and its targets ensures the data sent by the GVOC components always reach the intended target.

The second component(s) to come into play are the sensors: the SenEnv first, then the OtherNodes components. They are respectively responsible for the gathering of private and public information inside the node. A node can possess as many sensors as necessary. Each of them has a process B that initiates WRITE transactions to forward its collected information to the common target component named SAE. The SAE acts as a memory on which every sensor possesses a reserved space for its data with read access only. In other words, the memory space on the SAE is divided equally between all the sensors components. The interconnect component IC1 placed between the sensors and the SAE in the figure ensures the correct addressing of the memories areas by the different sensors during transactions as well as the prevention from overwriting of data by a sensor in its reserved memory area. This is achieved by the implementation through a linear mapping function for transactions addresses.

The LModel component is the third initiator component to come into play after the simulation starts. It is responsible for the evaluation of sensor data available inside the node, i.e, in the SAE memory, on the one hand. On the other hand, it is responsible for the initiation of the self-expressive behavior of the node. Thus, its has a process C1 which initiates read transactions to read the sensor data out of the SAE memory and let them be evaluated. Through its second process C2, the LModel component finally initiates WRITE transactions to forward the results or the necessary information to the SEE component to trigger the self-expression of the node, if necessary.

After the LModel follows the SEE component. As described in the Table I, the SEE component is a bridge component which embodies the self-expressive engine. Accordingly, it analyses the information previously received from the LModel component and selects the action to be taken. Following this, its process D starts WRITE transactions either to the actuators embodied here by the target component of the Model bearing the same name or to the external actuators embodied here by the target components named ExtActions or to both.

The monitor, which embodies the Monitor/Controller of the self-aware and self-expressive node, is here implemented in its simplest form, which is a monitor of the self-aware and self-expressive engines' actions. To this end, it has a process E1 which initiates READ transactions towards the LModel component to read the report data of its actions stored in its internal report memory. This occurs immediately after the evaluation of the sensor data in C1. It also has a process E2 which, similarly to E1, initiates WRITE transactions to read the report data in the report memory of the SEE component right after the actions in the node has been taken. In contrast to other components, the monitor actions must not be executed in every transaction cycle. According to its needs, the user has the possibility to give the period for the activation of the monitor actions.

After the SEE component or the monitor is activated, the LModel component operates again. It reads the next available data out of the SAE memory and the whole process described above from that point on is repeated. This occurs until all the sensor data stored on the SAE memory are evaluated. Then, the whole operation cycle, named here as **process-cycle**, starts again. Here, the number of process-cycles inside a node during the simulation of a model depends on the whole amount of sensor information to be processed inside the node and in each process-cycle. The latter is defined by the user before simulation starts.

All the above described TLM components of a self-aware and self-expressive node are encapsulated inside a SystemC module named "node" as shown in Figure 2. This module represents the highest hierarchy in the model and is responsible

 ISBN: 978-1-61208-341-4 

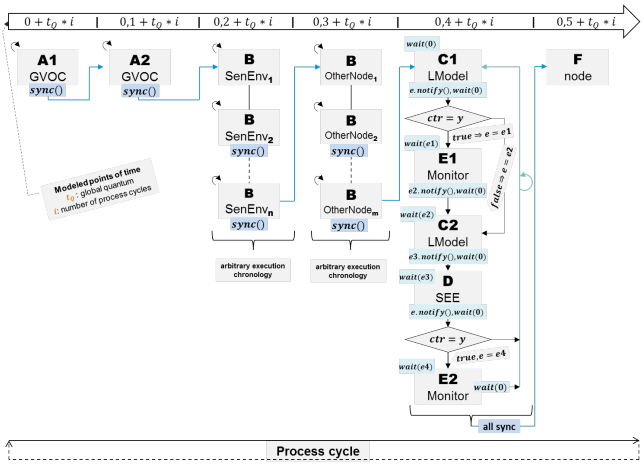


Figure 3. The implemented execution chronology of processes.

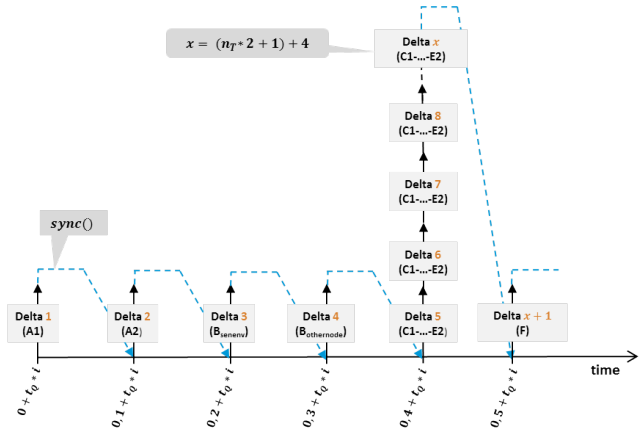


Figure 4. Delta-cycles within each process-cycle.

for the generation and instantiation of the TLM components inside each node according to the user specifications as well as for the resulting sockets and port-bindings for the communication among the components inside the node and between the system nodes at simulation start. Furthermore it has a process F that is the last to be executed in the process-cycle. Process F is just a synchronization process, i.e., it does not initiate any transactions, it is executed only once in each a process-cycle and ensures that processes in all node of a multi-node self-aware and self-expressive system end at the same simulation time in a process-cycle.

*2) Processes:*

From the model description above, it appears that there is a precise execution chronology of transactions and thus of processes in the environment that is a prerequisite and must always be maintain during simulation in order to reflect the functionality of a self-aware and self-expressive node prescribed by the architecture. This is: $\mathrm{A1} \rightarrow \mathrm{A2} \rightarrow \mathrm{B} \rightarrow \{\mathrm{C1} \rightarrow [\mathrm{E1}] \rightarrow \mathrm{C2} \rightarrow \mathrm{D} \rightarrow [\mathrm{E2}]\}$. It represents the so-called process-cycle previously mentioned. The processes within the curly brackets, here referred to as process chain, are executed alternately after each transaction until all sensor information available in the SAE memory are evaluated. Finally, the processes E1 and E2 of the monitor in the square bracket are activated in specific process-cycle intervals. A more detailed and precise view of the execution chronology of processes within a node is shown in Figure 3 and the following explain how this has been ensured through implementation.

For the temporal processes in a TLM Model, SystemC offers two possibilities [8]: The loosely-timed modeling style, which just models the start and end times of transactions but enables fast simulation times. The second one is the approximately-timed modeling style, which really details the phases of a transaction but at the cost of simulation performance. Giving the fact that the environment has to deal with industrial size systems, the simulation performance was a main concern during the implementation and has therefore lead us to the choice of loosely-timed modeling style. The Loosely timed modeling style implies the temporal decoupling of processes. This is implemented by means of the so called global time quantum $t_{globQ}$. The global time quantum is a time values that defines the synchronization (suspension) times of all processes $t_{sync}$. Each process run ahead simulation time and is executed as many times as possible till the next synchronization point is reached. The next synchronization point $t_{sync,next}$ of a process depends on the latency times of the transactions it executes after the previous synchronization and the actual simulation time $t_{sim}$. Within each process, the latency time $t_{trans\_delay}$ of each executed transaction is added up to the so-called local time offset $t_{off}$, which is then used to verify, if the process has to synchronize, i.e., be suspended. A suspended process can run again, only when the scheduler has advanced the simulation time $t_{sim}$ of this same local time offset. So, for every process of the model the following formulas always hold:

$$\text{synchronization points:} \quad t_{sync} = N * t_{globQ}, \quad N \in \mathbb{N} \quad (1)$$

$$\text{between 2 subsequent } t_{sync}: \sum_{i=1}^{n} t_{trans\_delay,i} = t_{off}, n \in \mathbb{N}^* \quad (2)$$

$$\text{synchronization condition:} \quad t_{off} \geq t_{sync,next} - t_{sim} \quad (3)$$

From the above described behavior of temporal decoupled processes and their formulas, it results that the execution time of a process after a synchronization relies on the following three key parameters, which can be modified during implementation: The first execution time at simulation start, the global quantum and the local time offset, which is the sum of the latency times of the transactions.

In order to fix this execution and obtain the precise execution chronology illustrated in Figure 3, we, therefore firstly chose the adequate first execution times of some processes. Secondly, we determined the value range for the global quantum. Third and finally, with the assumption that all transactions of a process always have the same latency time, we decided to let the user input the number of transactions to be executed per process-cycles and we established formulas for the automatic calculation of the latency times of the transactions between the synchronization points during the elaboration and the generation of the simulation model, such that the local time offset between two synchronization points always equals the global quantum.

- The first executions times $t_{beg}$:

To fix the first execution times of processes in our environment, we make use of time-outs. A time-out occurs when the method $wait(t)$ is called with a time-object $t \in \mathbb{Z}^+$ as parameter. When executed, the running process is suspended and resumed after the given time period has elapsed.

So, with respect to the prerequisite execution chronology of processes and independently of the time unit, we respectively chose the following times $t_{beg}$ for the processes A2, C1 and F: $0.1,\ 0.2, 0.5$. For the processes B's, for a better overview,

we chose $t_{beg} = 0.2$ respectively chose $t_{beg} = 0.3$ for the ones located inside the SenEnv components and $t_{beg} = 0.4$ for the others located in the OtherNodes components. This method call is executed only once at simulation start in each of these processes.

To achieve the alternate execution of processes in the process chain, we make use of time-outs and events. Each process of the chain notifies an event belonging to the dynamic sensitivity list of the next process and calls the method $wait()$ with $t = 0$ as parameter, after it has executed a transaction (see Figure 3). This produces a so-called delta-cycle, i.e., a process is suspended and resumed at the same simulation time, but in the next delta-cycle. So, the processing of a sensor data set and the reaction based upon the processing results always happens at the same simulation time but in different delta-cycles. And all sensor data set stored in a the SAE in a process-cycle are all processed by the process chain within the same process-cycle.

- The global time quantum $t_{globQ}$:

In [9], it is proved that the global time quantum of a TLM Model should be determined, in accordance with the whole simulation time period, so that the number of resulting synchronizations $n_{sync}$ or delta cycles $n_{delta_cycles}$ doesn't exceed a few hundred thousands. So the following inequalities should hold:

$$n_{sync} \leq 100000 \tag{4}$$

$$\text{or} \quad n_{delta\_cycles} \leq 100000 \tag{5}$$

Assuming that $t_{sim}$ denotes the whole simulation period, the number of synchronizations in the model can be calculated with the following formula:

$$n_{sync} = \left\lfloor \frac{t_{sim}}{t_{globQ}} \right\rfloor \tag{6}$$

In our model, the number of generated delta-cycles by the processes between two synchronization points is always the same and is illustrated in Figure 4. The temporal decoupled processes A1, A2, B's, and F always generate one delta-cycle. Because of the additional time-outs used in process chain to ensure the prerequisite alternate behavior, as described in the previous paragraphs, the process chain always produces two delta-cycles to complete the evaluation of a single sensor data set. Given that the process chain has to evaluate all sensor data set available on the SAE memory within a process-cycle, the number of generated delta cycles by the process chain therefore depends on the number of (read) transactions initiated by process C1 during a process-cycle. Finally there is an additional delta-cycle generated at the end of the process chain's execution for the synchronization of its processes. Thus, we have:

$$n_{delta\_cycles} = (N_T * 2 + 6) * n_{sync} * n_{nodes\_nr} \tag{7}$$

where $n_{nodes\_nr}$ is the number of nodes in the simulated system and $N_T$ is the number of (read) transactions of C1 between two subsequent synchronizations points. The formulas (4), (5), (6), (7) above lead us to the following formulas for the value range of the global time quantum:

$$\left\lfloor \frac{t_{sim}}{t_{globQ}} \right\rfloor \leq \frac{100000}{(N_T * 2 + 6) * n_{sync} * n_{nodes\_nr}} \tag{8}$$

- The latency times of transactions:

From the given synchronization condition (3) for temporally decoupled processes, it results that the global quantum is always less or equal to the sum of the latency times of all executed transactions between two synchronization points. Thus, with $t_{trans\_delay}$ denoting the latency time of the ith transaction of a process between two synchronizations points, we have:

$$t_{globQ} \leq \sum_{i=1}^{n} t_{trans\_delay,i} = t_{off} \tag{9}$$

Assuming that the value of local time offset is equal to the global quantum and that all the transactions between the synchronization points have the same latency times $t_{trans\_delay}$, we were able to derive the formulas below for the number of transactions of the processes between every two subsequent synchronization points:

$$t_{trans\_delay} = t_{globQ}/N_T \tag{10}$$

where $N_T$ denotes the number of transactions of each of the processes per process-cycle . This is given by the user before simulation start for the processes A1, A2 and B. Given the fact that process C1 and C2 have to read and evaluate all sensor information stored inside the node in a simulation cycle within the same simulation cycle, the number of transactions that they generate in each simulation cycle is equal to the mathematical product of the number of transactions $n_{T_B}$ generated by each sensor and the number of available sensors $n_s$ in the system. Thus,

$$N_{T_{C1,C2}} = n_{T_B} * n_s \tag{11}$$

This also applies the processes E1, E2 and D of the process chain, because they run in each simulation cycle as many times as the processes C1 and C2. Thus,

$$N_{T_{E1,E2,D}} = N_{T_{C1,C2}} \tag{12}$$

A transaction can be either a single transaction or a burst and the latency times of a single transaction $t_{single\_delay}$ differs from the latency times of a burst transaction, which is actually what formula (10) computes. The interrelation between both latency times is:

$$t_{trans\_delay} = t_{single\_delay} * BL \tag{13}$$

$$\text{with } BL = \left\lceil \frac{DL_{max}}{BUSWIDTH/8} \right\rceil$$

By substituting (11) in (10), we finally obtain the following formulas for the latency times of the single transactions for the processes A1, A2, B's

$$t_{single\_delay} = t_{globQ}/(n_{T_B} * BL) \tag{14}$$

and for the process chain:

$$t_{single\_delay} = t_{globQ}/(n_{T_B} * n_s * BL) \tag{15}$$

Applying the above computed formulas as well as the simulation-execution mechanisms as described above enabled us to meet our objective relative to the execution chronology of processes during simulation. For each process, we

$$t_i = t_{beg} + i * t_{globQ} \text{ mit } i = n - 1 \text{ und } i \in \mathbb{N} \tag{16}$$

where $i$ denotes the i-*th* process-cycle in the simulation.

## IV. USE CASE

For test and validation purposes, an avionic subsystem consisting of a single self-aware and self-expressive node with an alternative concept of fault-tolerance has been modeled and simulated using the introduced simulation and modeling environment. This concrete system is presented in the next section. Additionally the simulation results are presented and discussed.

### A. Scenario description

The system under investigation is an avionic subsystem on which an application composed of a safety relevant thread $C_r$ and two optional, i.e., not safety relevant, threads $O1$ and $O2$ is installed. The safety relevant thread is designed with triple modular redundancy [10] according to the reliability requirement standards [11]. This subsystem consists of a single self-aware and self-expressive node and the idea here is to make use of the self-aware and self-expressive capabilities of this node to drive fault tolerance and mitigation strategies.

In details, some physical properties of the system, here in our exemplary use case scenario the temperature, are measured by the sensors during service and compared with their known empirical values. Differently than in today's traditional fault tolerance designs, The 1st and 2nd copy of the critical thread, $C_r(1)$ and $C_r(2)$, are running at the system start. And its 3rd copy, $C_r(3)$ is only generated and turned on in case of discrepancy between the measured temperature values and the given empirical values of the temperature. At the same time, the optional threads are progressively shut down. Both, the generation and the turn-on procedure of the third copy of $C_r$ as well as the turn-off procedures of the optional threads happen progressively. The objective here is to secure the operation of the critical thread $C_r$, which execute the safety relevant operations of the subsystems. If the measured value of the temperature still hasn't fall back in the desired value range after the actions cited above have been taken, then the system is restarted and the threads are bring back to the start configuration. But, as soon as the temperature values measured by the sensors comply with the given empirical values, the optional threads are progressively turned on while the 3rd copy of the critical thread is progressively switched back and deleted.

An example of the modification of the threads' execution state according to the monitoring of the system temperature as described above is illustrated in Figure 5.

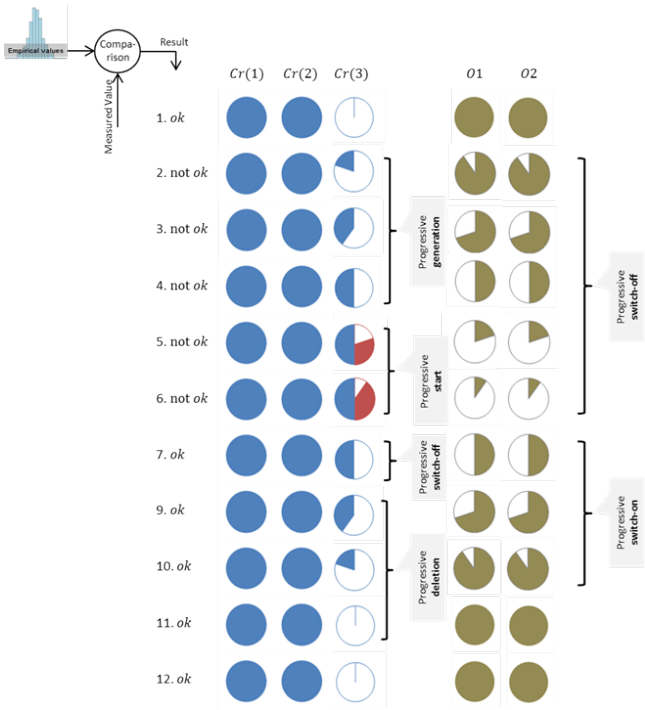


Figure 5. Example of the threads' execution state in the system according to the comparison results.

### B. Prototyp Building

*1) The threads:*

To model this concrete system, we implement the threads as classes with a constructor parameter of type *string* representing the type of the thread, here $typ = \{critical, optional\}$ and a value $s$ of type *float* representing the state of a thread, here $s \in [0, 1]$. During simulation, with respect to the results of the continuously comparison between the measured and empirical value of the temperature (self-awareness), this state variable is altered by the actuator of the node (self-expressive behavior) to model the behavior of the corresponding threads. For the optional threads, this indicates the progressive switch-on and off processes. For the critical thread Cr(3), it indicates its progressive generation, switch-on, switch-off as well as its deletion. For an optional thread, i.e, $typ = critical$, $s = 0$ means that the thread is switched off and $s = 1$ means that the thread is switched on or running. For a critical thread, $s = 0$ means that it is deleted, $s = 0.5$ means that it is generated and $s = 1$ means that it is switched on or running.

*2) The functionality of the components:*

- The SenEnv component

Due to the fact that, there is only one physical parameter to be observed, the system's own temperature, the model only needs one SenEnv component. The measured temperature values are given here in a matlab file. Thus, the SenEnv component reads out the file at program start and stores the data in a vector. In each process-cycle, the SenEnv component executes a transaction to transfer a single temperature value $t$ of the vector to the SAE component.

- The LModel component

The value previously stored in the SAE component by the sensor is read by the LModel and compared with the given maximal empirical value $T_{max}$ of the system temperature. Using the given frequency distribution of the temperature empirical values and depending on the comparison results, a value $v(t)$, which will help to initiate the self-expressive behavior of the system, is computed as follows:

$$v(t) = \begin{cases} -1.0 & \text{when } t > T_{max} \\ N[i] & \text{else} \end{cases} \quad \text{with } i = round(t - T_{max}) \tag{17}$$

The frequency distribution is given as a matlab file and stored at simulation start in the LModel component in a vector $N$. Here, $N[i]$ denotes the frequency density of this distribution. The computed value of $v(t)$ is thereupon transferred to the SEE component by the process C2 of LModel.

- The SEE component

This component uses the received value v(t) to compute a so-called decision vector $E(v)$. This vector consists of 5 values and represents the choice of the self-expressive engine of the node regarding the action to be performed on the threads

according to the node's awareness and reasoning.

$$E(v) = (\underbrace{e_1}_{Cr(1)} \quad \underbrace{e_2}_{Cr(2)} \quad \underbrace{e_3}_{Cr(3)} \quad \underbrace{e_4}_{O1} \quad \underbrace{e_5}_{O2}) \tag{18}$$

with $e_i \in [-1, 1]$and $i \in \{1, 2, 3, 4, 5\}$. So, each element of this vector E is the update value to be used by the actuator to alter the state of the corresponding thread of the system. In case that the system must restart, the SEE component will compute the following decision vector: $E(v) = \{-1, -1, -1, -1, -1\}$, else the following formula is used:

$$\begin{aligned} e_i(v) &= \begin{cases} -0.01 & \text{with } v = -1 \\ v - N_{max} & \text{with } v \leq 0.9 * N_{max} \\ v - N_{max} + 0.1 & \text{else} \end{cases} \\ e_3(v) &= -1 * e_i(v) \end{aligned} \tag{19}$$

with $i \in \{4, 5\}$ and $N_{max}$ maximal value of the frequency density. After computation, the SEE component finally forwards this vector to the actuator of the node.

- The actuator component

As already mentioned above, the actuator component uses the received decision vector $E(v)$ to update the threads execution state according to the nodes self-awareness. So, for each of the threads $i$ we have the following:

$$s_{new,i} = s_{actual,i} + e_i. \tag{20}$$

During the simulation, the computed values in each process-cycle are stored in a matlab file.

- The monitor component

Here, the task of the monitor is to read the report data of the SAE and SEE components and to monitor them, i.e., to write them in a matlab file that is used to control the values $v(t)$ and $E(v)$ computed respectively by the SAE and the SEE components.

- The GVOC component

The only constraint given here is the maximum value $T_{max}$ for the system temperature. So, in each process-cycle, GVOC just forwards $T_{max}$ to the SEE and Monitor components.

- The OtherNode and the ExtActions components

As we have mentioned above, this system under investigation comprises a single node. Thus, this prototype don't need any OtherNode component. In addition, there is no actions to be performed on the node's environment. This implies that there is also no need for ExtActions components in this prototype.

### C. Simulation and Evaluation

With a total of 2500 given temperature values to be processed inside the self-aware and self-expressive node, the number of temperature value to be process within a process-cycle set to one, i.e., one transaction of the sensor per process-cycle, a simulation time period $t_{sim} = 5002ms$ and a global quantum time of $2ms$ were chosen according to (8) derived in section 3. Our model needed exactly 17500 transactions for all processes, 2001 synchronisations and lasted around 3 minutes. A total of 16008 delta-cycles were generated, which agrees with (7).

Figure 6 displays on the top line chart, the run of the given measured temperature values and, on the bottom line chart, the run of the threads states values computed over the whole simulation period according to the temperature values. Here, the maximal empirical temperature value is $T_{max} = 87\,^{\circ}\mathrm{C}$. One can realize from this illustration that the temperature of

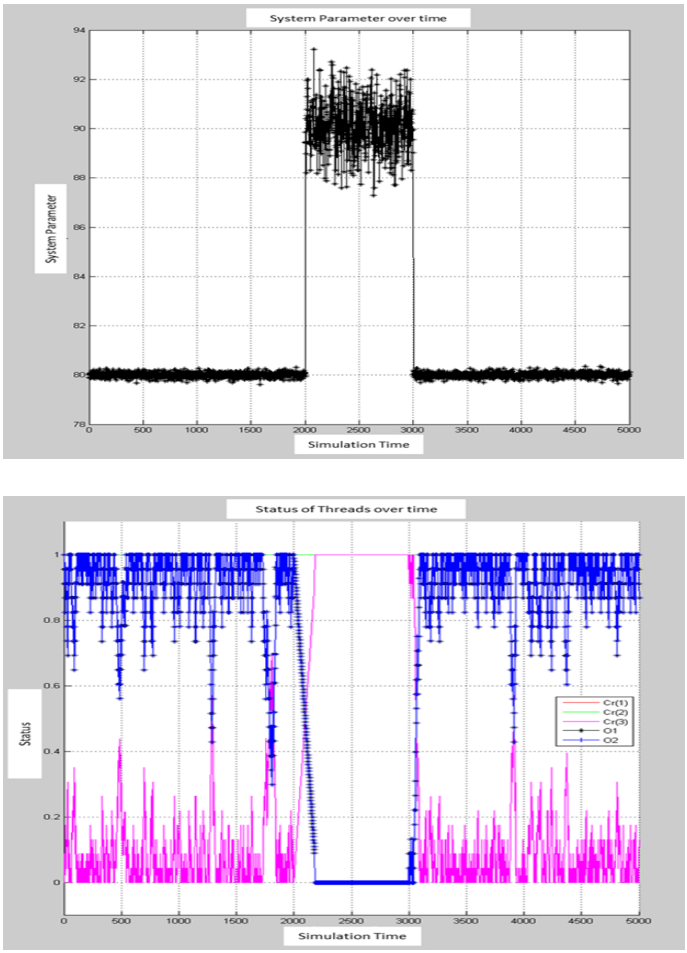


Figure 6. Temperature and threads execution state during the simulation period.

the system is not constant. But the first and last thousand values remain below the given maximum $T_{max}$ while the remaining five hundred values exceed it. As expected, the state values $s$ of the threads vary accordingly over the simulation period. Indeed, the state value of $Cr(3)$, the 3rd copy of the critical thread, is also not constant but always remains under the value $0.5$ when the temperature isn't to near to the given maximum. When the temperature continues to rise, approaches, reaches or exceeds the given maximum, $Cr(3)$ is generated ($s_{Cr(3)} = 0.5$), progressively switched on and remains in this state ($s_{Cr(3)} = 1$). Meanwhile, the optional threads are switched off ($s_{O1,O2} = 0$). As soon as the temperature falls back, the state value of $s_{Cr(3)}$ decreases while $s_{O1}$ and $s_{O2}$ increase. Some specific points of the simulation have been captured in Figure 7 and underpin the above statement.

## V. Conclusion

In this paper, we described a reference architectural framework developed to structure the requirements for the design of computing system with self-aware and self-expressive behaviour. Subsequently, we presented a modelling and simulation environment developed in SystemC using the Transaction-Level Modelling (TLM) and based on the previous mentioned framework for the construction of prototype and the tests and validation of novel concepts developed based on both properties. The presented environment can be used to build virtual prototypes of self-aware and self-expressive systems for industrial systems. Moreover, through the clear separation between

 ISBN: 978-1-61208-341-4

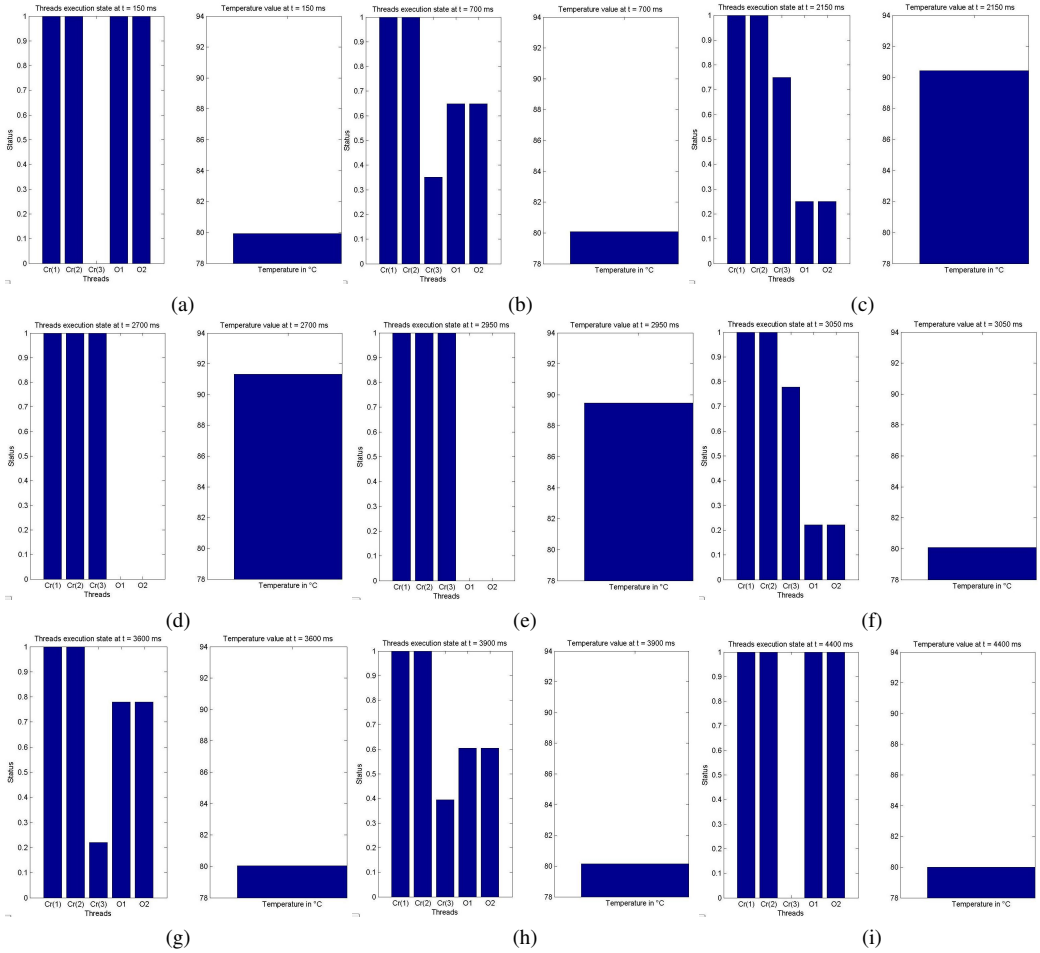


Figure 7. Threads's execution state according to the temperature at specific simulation points.

the proper components' functionalities and the communication among them on the one hand, the implemented accurate and reliable execution chronology of temporal decoupled processes used to encapsulate them, the environment achieves fine timing resolutions and ensures the functionality described in the reference architecture. As the third and final part of this paper, we presented the model was used to build a prototype of a single-node self-aware and self-expressive system presenting a novel concept for fault-tolerance in avionics systems could be built with the model using the test environment. The simulation results have also been presented and discussed.

Ongoing work is devoted to the optimization of the implementation of the presented environment and the development of more novel fault-tolerance concepts and approaches that are better suitable to the next generation of computing systems, systems with self-properties, and can lessen the performance and functionality restraining high redundancy level of safety-critical systems, most particularly of avionics embedded systems.

## ACKNOWLEDGMENT

The research leading to these results has received funding from the European Union Seventh Framework Program under grant agreement no 257906.

## REFERENCES

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, 2003, pp. 41–50.

[2] epics, "EPiCS Project," Jan 2014. [Online]. Available: http://www.epics-project.eu/

[3] sapere, "SAPERE Project," Mar 2014. [Online]. Available: http://www.sapere-project.eu/

[4] recognition, "Recognition Project," Mar 2014. [Online]. Available: http://www.recognition-project.eu/

[5] S. Parsons, R. Bahsoon, P. R. Lewis, and X. Yao, "Towards a better understanding of self-awareness and self-expression within software systems," University of Birmingham, School of Computer Science, UK, Tech. Rep. CSR-11-03, Apr 2011.

[6] P. R. Lewis et al., "A survey of self-awareness and its application in computing systems," in Proc. Int. Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW). IEEE Computer Society, 2011, pp. 102–107.

[7] T. Becker et al., "EPiCS: Engineering proprioception in computing systems," in Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on, 2012, pp. 353–360.

[8] I. C. Society, IEEE Standard for Standard SystemC Language Reference Manual - IEEE Std 1666$^{TM}$-2011, 2012.

[9] F. Kesel, Modeling of digital Systems with SystemC: From the RTL- to the Transaction-Level-Modeling. Oldenbourg Wissenschaftsverlag, 2012.

[10] R. Orsagh, D. Brown, P. Kalgren, A. Byington, C.S. ; Hess, and T. Dabney, "Prognostic health management for avionic systems," in Aerospace Conference, IEEE , 2006, pp. 1213–1219.

[11] M. Pignol, "COTS-based applications in space avionics," in DATE 2010, 2010, pp. 1213–1219.

 ISBN: 978-1-61208-341-4 