View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aston Publications Explorer

A Modelling and Simulation Environment for Self-aware and Self-expressive Systems

Tatiana Djaba Nya, Stephan C. Stilkerich
EADS Innovation Works
Munich, Germany
email: {tatiana.djabanya, stephan.stilkerich}@eads.net

Peter R. Lewis
CERCIA, School of Computer Science
University of Birmingham, UK
email: p.r.lewis@cs.bham.ac.uk

***Abstract*—Self-awareness and self-expression are promising architectural concepts for embedded systems to be equipped with to match them with dedicated application scenarios and constraints in the avionic and space-flight industry. Typically, these systems operate in largely undefined environments and are not reachable after deployment for a long time or even never ever again. This paper introduces a reference architecture as well as a novel modelling and simulation environment for self-aware and self-expressive systems with transaction level modelling, simulation and detailed modelling capabilities for hardware aspects, precise process chronology execution as well as fine timing resolutions. Furthermore, industrial relevant system sizes with several self-aware and self-expressive nodes can be handled by the modelling and simulation environment.**

I. Introduction

Self-awareness and self-expression are promising architectural concepts for embedded systems to be equipped with to match them with dedicated application scenarios and constraints in the avionic and space-flight industry. Systems that profit from these kind of self-aware and self-expressive capabilities are (1) *avionic systems*, (2) *autonomous flying systems*, (3) *special satellites*, (4) *deep-space mission systems* and (5) *exploratory space mission systems*. Typically, these systems are highly dependable, represent a substantial investment, operate in largely undefined and changing environments that are impossible to define during system design, and are not reachable after deployment for a long time or even ever again [1]. Consequently, it is today's industrial practice to tremendously over-design these systems with respect to redundancy, diverse equipment, long operationally proven components and static breakdown mitigation concepts. Furthermore, all of these systems have stringent weight, power, size and density constraints [2]. This significantly limits the overall processing performance and the kinds of implementable functionality. Systems with self-aware and self-expressive features can overcome some of these limitations and offer a completely new architectural concept to deal with unpredictable environments through flexible, not pre-defined, sub-system adaptation.

Between research at the conceptional level of self-aware and self-expressive systems and the level of implementing first proof-of-concept demonstrators, there is a gap to systematically model, simulate and study the behaviour and performance of particular self-aware and self-expressive systems. This gap is filled by our novel modelling and simulation environment for self-aware and self-expressive systems. It offers modularised modelling, transaction level abstraction and execution, possibilities for detailed hardware modelling, fine resolution timing and industrial relevant system handling.

This paper is organised as follows: the next section describes our self-aware and self-expressive reference architecture that serves as theoretical basis for the implementation of the modelling and simulation environment. Section III introduces the novel simulation environment that is based on our reference architecture. All components of the simulation environment, their dependencies and interconnections are explained. The execution chronology, simulation time offsets and transaction flows are described. Finally section IV comments on the utilisation of the novel simulation environment for avionic and space-flight applications, while section V concludes this paper.

II. Reference Architecture

System self-awareness, and adaptive behaviour based on it, have long been recognised as enablers for advanced autonomic behaviour [3]. Appropriate levels and forms of self-awareness will be essential for systems, which operate for long run times, in unpredictable and changing environments with minimal human intervention, while being heavily resource constrained. Informally, we consider self-awareness to be concerned with the acquisition and representation of knowledge about a system by that system. The complementary concept of self-expression then describes behaviour based on a system's self-awareness [4]. In order to lay the foundations for systems that possess self-awareness and self-expression capabilities, concepts from psychology and cognitive science are being reinterpreted in a computational context [4], [5]. This approach has also been followed to develop other nature-inspired computing paradigms, which have enjoyed great success in a wide range of practical applications, despite the frequent lack of the assurances usually required for engineered systems.

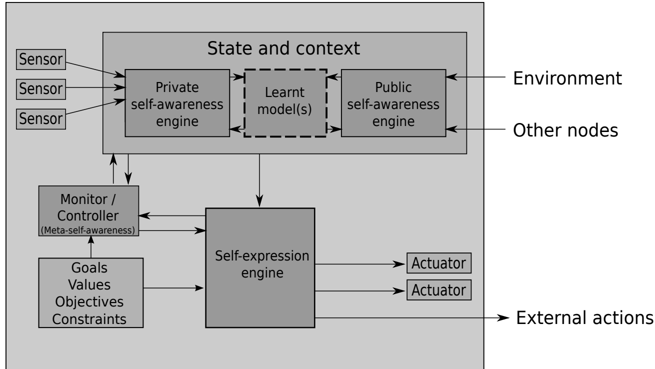

Figure 1. Reference architecture for a single self-aware and self-expressive node.

A. Self-aware, Self-expressive Nodes

As part of this effort a reference architectural framework for a self-aware, self-expressive *node* was developed [6]. This is shown in figure 1, and defines the high level structure of such a node and its required conceptual components. Such a *node* need not to be a specific physical system, but instead provides a conceptual container for the system being considered: the element in that context which is being referred to as *self*. A node could therefore be, for example, an autonomous agent, a running thread, a physical machine or a collective of these. Importantly, the node represents a level of abstraction at which knowledge acquisition, representation and behavioural processes occur.

The architecture deals with the concept of public and private self-awareness [4], by specifying conceptual components for building knowledge from both internal and external sources of information. Additionally, a meta-self-awareness [4] component is identified, responsible for knowledge concerning the system's own self-awareness and self-expression processes, and for adapting them as necessary.

B. Online Learning

Due to the unpredictability associated with both deployment environments and the dynamics within them, one key challenge in realising self-awareness and self-expression in computing systems is the appropriate use of effective online learning schemes. Online learning is applied in two contexts within a self-aware, self-expressive system: at the adaptation level and at the meta level [6]. In the reference architectural framework, online learning algorithms instantiate two conceptual components at the adaptation level:

- the self-awareness engines, where sensor data is collected, analysed and, if appropriate, knowledge obtained from it is represented.
- the self-expression engine, where behavioural learning (e.g. action selection and strategy selection) takes place.

Additionally, online learning at the meta level occurs in the meta-self-awareness component, where models of the node's own behaviour are built online, and acted upon.

C. Approach

In attempting to draw general conclusions about the benefits of self-awareness and self-expression for computing and engineering, we turn to our understanding of these concepts in psychology, as well as previous efforts to apply them to computing [4]. A key finding of that survey was that the term self-awareness has been used in a variety of ways within computer science and engineering literature. The general concept of self-aware computing covers but is not limited to all of these cases. Our approach is therefore to design systems based on the flexible node architectural framework, which defines how self-awareness, self-expression and meta-self-awareness concepts can be combined to achieve run time learning and adaptation. This reference architectural framework allows for a wide range of approaches to be taken, while ensuring coherence between the three key activities of self-awareness, self-expression and meta-self-awareness.

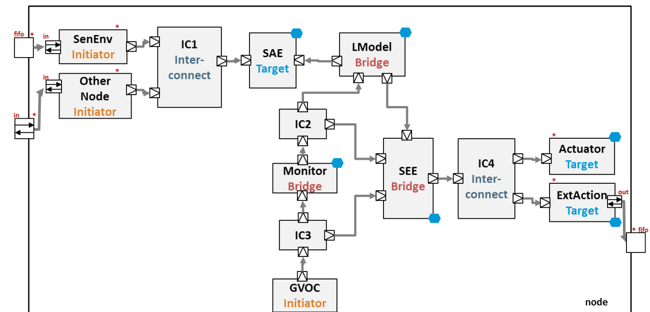

Figure 2. Structure of the simulation environment.

III. Simulation Environment

The reference architecture introduced in section II serves as basis for the detailed component implementation of the modelling and simulation environment. For the general modelling and simulation methodology, features such as overall simulation performance, scalability and abstraction levels at which to study self-aware and self-expressive systems have led to the decision of utilising a *Transaction Level modelling (TLM)* approach. A TLM approach typically separates details of the communication among modules from the details of the implementation of the modules and details of the overall communication architecture. In compliance with the defined reference architecture, the following TLM components as well as their roles were identified for the self-aware and self-expressive model (cf. figure 2):

- SenEnv: This TLM component represents the internal and the external sensor environment of the node. Each node can possess at least one or as many as required SenEnv modules, depending on the concrete system.
- OtherNodes: Receives information from other self-aware and self-expressive nodes in a self-aware and self-expressive system. A particular node can possess as many OtherNodes modules as required.

- SAE: Together with LMODEL, this represents the self-awareness functionality. SAE collects and stores the information sent by all the internal and external sensors.
- LMODEL: This component represents the part of the self-awareness functionality, which processes the received sensor information. The separation of the self-awareness functionality into two components was necessary to coordinate the transactions and overall synchronisation.
- GVOC: This component embodies the predefined Goals, Values, Objectives and Constraints of the self-aware and self-expressive node.
- MONITOR: The monitor component controls the self-awareness and self-expressive components. It has the system rights to intervene in the node processing to redirect or refocus lower level of self-awareness and self-expression in the node.
- SEE: A component representing the self-expression engine, this component will take decisions about what kind of actions to take, according to the received data from the self-aware engine component complex and in detail data from LMODEL, simultaneously taking GVOC data into account.
- ACTUATOR: Represents the internal actuator(s) of the node. This component is the target of SEE and each self-aware and self-expressive node can have several actuators, depending on the concrete system under investigation.
- EXTACTION: This component represents an actuator that has access to the external environment of a self-aware and self-expressive node and is, like the actuator component, the target of the self-expressive engine during transactions.
- IC1-4: These interconnect components of the model (cf. figure 2) are used where several initiators communicate with the same target or an initiator communicates with several targets over the same transaction type.

Additionally to the described TLM components, there is a high-level module, which coordinates the initial generation of the components, the port and socket bindings of a particular node as well as all of these generation processes for systems with several self-aware and self-expressive nodes.

A. Processes

In order to describe the detailed functional behaviour of a TLM component as well as to define the transaction level behaviour of a component, our approach utilises processes to encapsulate these actions. The following sections present the existing processes in our proposed modelling and simulation environment as well as the implementation of the temporal decoupling of these processes in the model.

1) Processes A1 and A2: A1 and A2 are located in the initiator component GVOC, as shown in figure 3. They transfer the specified goals, values, objectives and constraints

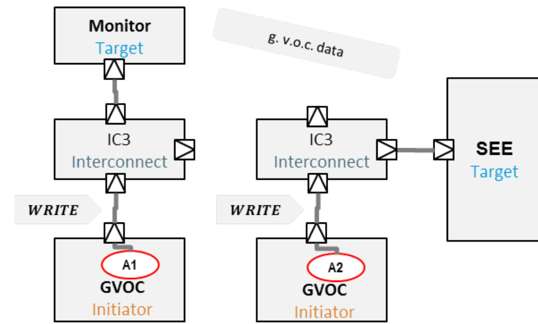

Figure 3. Transaction groups for the processes A1 (left) and A2 (right WRITE indicates that the transactions initiated here are write transactions

into the node by calling the blocking transport method. There is a transaction object for each of both processes. Process A1 transfer data to the MONITOR and process A2 to the component SEE. A1 and A2 are always the first processes to be executed in each system-simulation cycle and they always execute without any suspension till their respective next synchronisation points.

2) Process B: B is located in each of the sensor components SENENV, as shown in figure 4, and OTHERNODE. By means of the blocking transport method the data is transferred to SAE through the interconnect component IC1. The memory area of SAE is equally shared by the sensors in each node and the detailed address translation is always done by the interconnect component IC1 for each incoming method call and before the call is forwarded. All B processes run next after the A1 and A2 processes in every node and every simulation cycle, and also ahead simulation time until they reach their next synchronisation point.

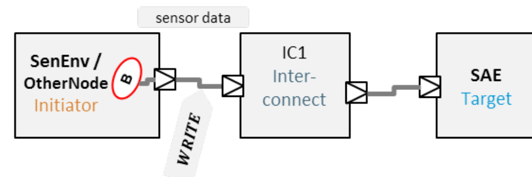

Figure 4. Transaction group for processes B - WRITE indicates that the initiated transactions are write transactions

3) Process C1 and C2: These processes are embedded in the LMODEL (figure 5). C1 is responsible for the linear readout of the memory space of SAE and the forwarding of that data into the module for processing. After the processing, LMODEL decides whether some measures have to be taken in view of the processing results. Its decision is finally sent to SEE by C2. C1 will always be executed before C2 in each node, during each simulation cycle and after all B processes were suspended. At every execution, process C1 creates an immediate notification and an delta notification. With the delta notification, process C1 is suspended and with the immediate notification an event belonging to the dynamic sensitivity list of process C2 is notified. Hence, C2 runs immediately after C1 is suspended.

C2 generates a transaction to transfer the results of LMODEL to SEE. Additionally, C2 executes an immediate notification to inform SEE after the last transaction is completed. The notified event belongs to the dynamic sensitivity list of process D, which is next on the set of executable processes.

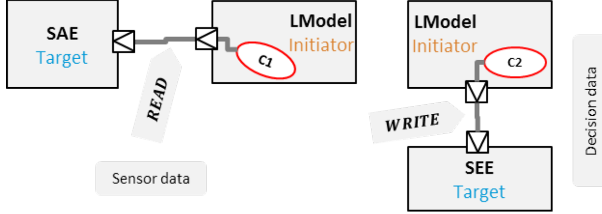

Figure 5. Transaction groups for processes C1 (left) and C2 (right). WRITE and READ indicate the type of transactions generated.

4) Process D: During each execution of D, which is located in SEE (figure 6), the data received from the component LMODEL is read out and evaluated for self-expressive actions. The dedicated self-expressive action is realised by process D through initiated write transactions to the corresponding actuators, EXTACTION or ACTUATOR.

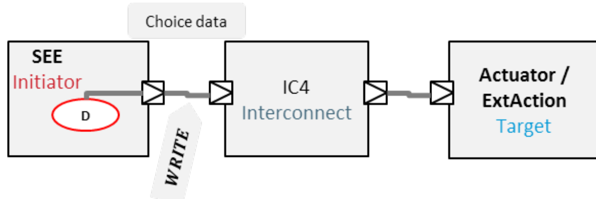

Figure 6. Transaction group of process D

5) Process E1 and E2: Both Processes E1 and E2 are located in the Monitor (figure 7). E1 is responsible for LMODEL and activated in each node immediately after process C1. E2 is activated directly after process D and is linked with SEE. In view of the fact that the monitor of the self-aware and self-expressive node doesn't have to constantly monitor the self-expressive and the self-awareness engines, these processes are activated at each simulation cycle but only executed at predefined intervals (counter) of simulation cycles. In case that the counter value is equal to the specified number, events belonging to the dynamic sensitivity list of process E1 and E2 are notified and the counter is reset. The immediate notification, which activates process E1, is executed by process C1 and the one which activates process E2, is executed in D. At each execution, process E1 reads the report memory of LMODEL and E2 reads the report memory of SEE.

6) Synchronisation process F: This process is embedded in the high-level module of the model. F runs in each simulation cycle only once and synchronises always immediately after it has started. The purpose of this process is to ensure that processes in all nodes of a multi-node self-aware and self-expressive system end at the same simulation time of

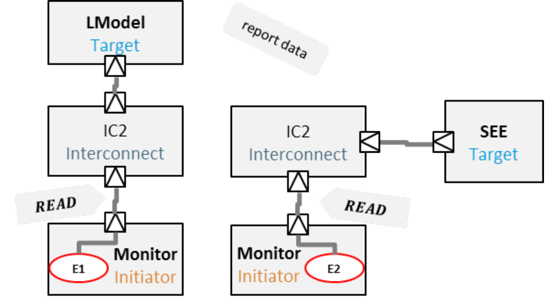

Figure 7. Transaction groups for processes E1 (left) and E2 (right). READ indicates that the generated transactions are read transactions

a cycle. This means that the control is yielded back to the simulation kernel only after all the processes (A - F) of all instantiated nodes have reached their next synchronisation point.

B. Execution chronology

From the above behavioural descriptions of the processes, it arises that the processes of a node always run in the same chronology, cf. figure 8, within a simulation cycle:

$$A1, A2 \rightarrow B \rightarrow \{C1 \rightarrow [E1] \rightarrow C2 \rightarrow [E2] \rightarrow D\} \rightarrow F.$$

Here, processes E1 and E2 run only in specific intervals of simulation cycles, which must be defined by the user before the simulation start. For processes in brackets, hereinafter referred to as process chain, they run alternately after each transaction until they reach their next synchronisation point.

C. Local time offset

Temporally decoupled processes [7] run always ahead of the simulation time and need to be suspended for a defined period of time, namely the local time offset t_{off}. A suspended process can run again, only when the scheduler has advanced the simulation time of this same local time offset t_{off}. Based on that fact, we can deduce that the next execution time $t_{sim,next}$ of a suspended temporally decoupled process always results from the sum of the actual simulation time and the local time offset of this process ($t_{sim,next} = t_{sim,actual} + t_{off}$).

The local time offset, in turn, results from the sum of latency times of all the transactions generated by a process between two synchronisation points. In our model, the latency times and the number of transactions between the synchronisation points of a process are automatically calculated during elaboration and generation of the simulation model, such that the local time offset between two synchronisation points always equals the global quantum, the point of process synchronisation. Therefore, the next execution time of a process after his last synchronisation always results from the sum of the actual simulation time and the specified global quantum $t_{glob_quantum}$. Hence we have $t_{sim,next} = t_{sim,actual} + t_{glob_quantum}$. From the known simulation time of the first execution of each process (t_{beg})

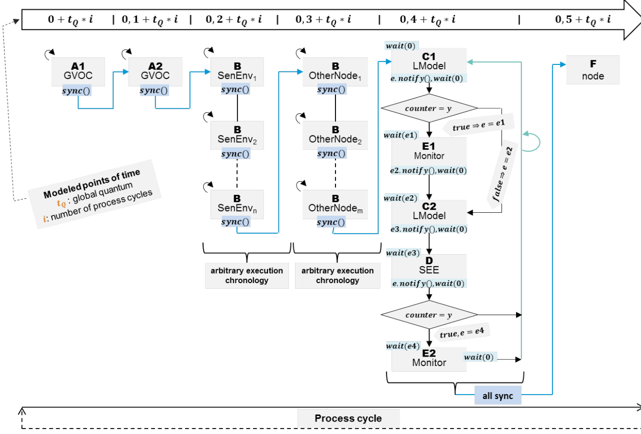

Figure 8. Implemented chronology of processes

and the above formula, we have the following starting times for the execution of a process
in the 1st simulation cycle with $t_0 = t_{beg}$ and in the 2nd simulation cycle with $t_1 = t_{beg} + t_{glob_quantum}$. Summarising, we have the following start time in the nth simulation cycle with $t_{n-1} = t_{beg} + (n-1) * t_{glob_quantum}$, with $n \in N^+$. With the formula above, we were able to derive the following general formula for the execution start time t_i of a process in every process cycle n: with $t_i = t_{beg} + i * t_{glob_quantum}$; $i = n - 1$, $n \in N^+$.

1) Global quantum: When using temporal decoupling, it is recommended that each process uses the same global time quantum. It should be determined, in accordance with the whole simulation time period, so that the number of resulting synchronisations or delta cycles doesn't exceed a few hundred thousands (see [8, p. 279]). Let's assume that t_{sim} denotes the whole simulation time period, n_{delta_cycles} the number of delta cycles and n_{sync} the number of synchronisations. The number of synchronisations in the model can be calculated with the following formula:

$$n_{sync} = \left\lfloor \frac{t_{sim}}{t_{glob_quantum}} \right\rfloor$$

Using the previous formula for the number of synchronisations, we can precisely derive the formula for the number of delta cycles in the model:

$$\begin{aligned} n_{delta_cycles} &= (n_T * 2 + 6) * n_{sync} * n_{nodes_nr} \\ &= (n_T * 2 + 6) * \left\lfloor \frac{t_{sim}}{t_{glob_quantum}} \right\rfloor * n_{nodes_nr} \end{aligned}$$

where $n_{nodes_nr} \geq 1$ denotes the number of nodes in the simulated system.

Thus, the following inequality holds:

$$\begin{aligned} n_{sync} &\leq 100000 \quad \Leftrightarrow \\ \left\lfloor \frac{t_{sim}}{t_{glob_quantum}} \right\rfloor &\leq 100000 \end{aligned}$$

or

$$\begin{aligned} n_{delta_cycles} &\leq 100000 \quad \Leftrightarrow \\ \left\lfloor \frac{t_{sim}}{t_{glob_quantum}} \right\rfloor * (6 + n_T * 2) * n_{nodes_nr} &\leq 100000 \end{aligned}$$

Both inequalities lead us to the following formula for the global quantum:

$$\left\lfloor \frac{\boldsymbol{t_{sim}}}{\boldsymbol{t_{glob_quantum}}} \right\rfloor \leq \frac{\mathbf{100000}}{\boldsymbol{(6 + n_T * 2) * n_{nodes_nr}}}$$

that is used in our proposed modelling and simulation environment.

2) Latency times of the transactions: From the synchronisation condition 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 synchronisation points. Thus:

$$t_{glob_quantum} \leq \sum_{i=1}^{n_T} t_{trans_delay,i} = t_{off}$$

where $t_{glob_quantum}$ global quantum, n_T number of transactions, $t_{trans_delay,i}$ latency of the ith transaction and t_{off} local time offset.

To determine the execution start times of the processes in the simulation cycles, we assumed that the value of local time offset is equal to the global quantum; $t_{glob_quantum} = t_{off}$.

Assuming that every transaction has the same latency, the sum of the latency times in the inequality above can be substituted by the product obtained when multiplying the latency of one transaction by the number of executed transactions between two synchronisation points. We have $t_{glob_quantum} \leq n_T * t_{trans_delay}$, where t_{trans_delay} denotes the latency of each transaction.

We can now derive the latency of each transaction generated by the processes A1, A2 and B's by $t_{trans_delay} = t_{glob_quantum}/n_T$, where n_T denotes the number of transaction per process cycle.

All sensor data stored in SAE has to be read and evaluated by LMODEL within a simulation cycle. The number of transactions generated by each of the processes C1 and C2 is given by $n_{T_{C1,C2}} = n_{T_B} * n_S$, where $n_{T_{C1,C2}}$: number of transactions generated in each process cycle by C1 and C2, n_{T_B}: number of transactions generated in each process cycle by each process B and n_S: the number of sensors in the model.

The same formula applies to 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.

By substituting the number of transactions in the previously derived formula for the latency of processes A1, A2 and B's, we obtain the following formula for the latency of each process in the process chain $t_{trans_delay} = t_{glob_quantum}/(n_{T_B} * n_S)$.

The transaction latency times that we are actually looking for are the latency times t_{single_delay} of the single transactions. These are not always equal to the latency times determined previously, precisely when the generated transactions are bursts, i.e. when the burst length is greater than one ($BL > 1$) ([8, p. 217]).

So we need the following formula showing the functional interrelation between the latency of a transaction and the latency of a single transaction in order to derive the searched formula:

$$t_{trans_delay} = t_{single_delay} * BL;\ BL = \left\lceil \frac{DL_{max}}{(BUSbits/8)} \right\rceil$$

This leads to the following formula for the latency of the single transactions in the node for the processes A1, A2 and B's as $t_{single_delay} = t_{glob_quantum}/(n_{T_B} * BL)$ and for the processes C1, C2, E1, E2 and D of the process chain as $t_{single_delay} = t_{glob_quantum}/(n_{T_B} * n_S * BL)$.

These are the formulas used in the model to automatically calculate the latency times of the single transactions in each node of a simulated system. This ensures the preconditioned execution chronology of the processes and transactions in each simulation cycle as well as the respective execution start times of the processes that we defined. All together, this realises a functional and timing correct simulation of self-aware and self-expressive systems.

IV. Evaluation

Both main parts of our advocated approach, the reference architecture (section II) as well as the modelling and simulation environment (section III), ran through intensive testing and refinement phases to reach the current stable version.

During a first testing and refinement phase of the modelling and simulation environment several more generic self-aware and self-expressive systems composed of just one node or composed of several nodes have been utilised to optimize the simulation performance, the execution chronology and the transaction and event handling of the different processes. Additionally, the data read, storage and transport mechanisms have been tested, bug-fixed and optimized to provide an environment that can handle industrial relevant systems in simulation run times of a few hours for data input sizes of several tens of thousands of input samples.

Beside the modelling and simulation work with these generic systems to test out and optimize the implementation, we currently use the environment for a self-aware and self-expressive system that realises novel concepts for fault tolerance to overcome one drawback of today's over-designed avionic systems. This concrete system is composed of one single self-aware and self-expressive node representing an avionic sub-system with safety critical functionality. Later extensions will realise several of these nodes that exchange information to guide the self-aware and self-expressive behaviour locally at one node and at all system nodes, simultaneously. Due to running patent applications further details are not possible.

V. Conclusion

In this paper we have presented a comprehensive approach for self-aware and self-expressive systems, including a systematic derivation of a reference architecture and a complete modelling and simulation environment that is based on that generic architectural template. With the defined reference architecture and the modelling and simulation environment at hand, we are able to systematically define, simulate and analyse systems with self-aware and self-expressive capabilities. Initial tests with multi-node systems and first results of a running project on alternative fault tolerance avionic concepts with one node proof our approach and underpin the industrial relevance of the implemented environment.

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

References

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

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

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

[4] P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, and X. Yao, "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, p. 102107.

[5] J. Schaumeier, J. Pitt, and G. Cabri, "A tripartite analytic framework for characterising awareness and self-awareness in autonomic systems research," in *Proc. Int. Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW).* IEEE Computer Society, 2012, pp. 157–162.

[6] T. Becker, A. Agne, P. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. Jensenius, and S. Stilkerich, "EPiCS: Engineering proprioception in computing systems," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, 2012, pp. 353–360.

[7] J. Fitchl, "A loosely coupled parallel LISP execution system," in *International Specialist Seminar on the Design and Application of Parallel Digital Processors*, 1988, pp. 128–133.

[8] F. Kesel, *Modellierung von digitalen Systemen mit SystemC: Von der RTL- zur Transaction-Level-Modellierung*. Oldenbourg Wissenschaftsverlag, 2012. [Online]. Available: http://books.google.de/books?id=ADxjNz0inTsC