Abstract

The architecture of modern compute systems become increasingly complex, heterogeneous and distributed. At the same time, functionality and performance are no longer the only aspects considered during the development of applications for such systems. Other aspects such as flexibility, power efficiency, resource usage, reliability and costs continue to grow in importance. The EPICS project (*Engineering Proprioception in Computing Systems*) aims to address and consider all these aspects by studying the concepts of „self-awareness“ and „self-expression“ and their integration into compute systems. Self-awareness enables any system and any application to collect and maintain different types of information about its internal state and its environment, and to reason about their behaviour. Self-expression enables the system to adapt its behaviour autonomously to changing conditions and requirements. In addition to these two properties, self-verification and fault tolerance concepts will be integrated to ensure the reliability and accuracy of a system after a self-adaptation. A compute system with all the properties cited above are named proprioceptive compute systems and represents a collection of proprioceptive nodes within the EPICS project.

For the illustration and analysis as well as for the optimization and validation of all the explored and integrated concepts, test experiments are indispensable. Unfortunately, these can not always be conducted on a real-time system for cost, flexibility and time reasons.

The aim of this work is, firstly, to provide the project a test environment and, secondly, to support the analysis and optimization of the integrated concepts of fault tolerance at the thread level (TLFT, *Thread Level Fault Tolerance*). For these purposes we developed here a model for proprioceptive nodes based on their reference architectural framework developed within the EPiCS project. The developed model is freely extensible and configurable, and is implemented at the transaction level (TLM, *Transaction Level Modelling*) in SystemC within the deployment environment Microsoft Visual Studio 2012. The model said was then extended by some hardware threads that are relevant for an aeronautical application. Thus, we obtained a simulator on which we conducted appropriate and consistent tests in regard to the analysis of the integrated thread-level fault tolerance. The results of this simulation were finally presented and interpreted.

Contents

1 Proprioceptive computing nodes **6**

1.1 Definition and properties of proprioceptive computing nodes 6

1.2 Reference architecture of a proprioceptive node 7

1.2.1 Self-aware node 8

1.2.2 Self-expressive node 9

1.2.3 Monitor/Controller 10

2 Modelling of the proprioceptive computing node **12**

2.1 Preliminary decisions 12

2.2 Identification of the TLM-components and their roles 13

2.3 Modelling of the communication systems 16

2.3.1 Interfaces and Sockets 17

2.3.2 Ports and channels 18

2.4 Processes and transactions 21

2.4.1 Definition of some important terms 21

2.4.2 Temporal decoupling 22

2.4.3 Behaviour descriptions 23

2.4.4 Execution chronology 28

2.4.5 Key parameters 33

2.4.6 Transaction flows 39

Bibliography **40**

List of Figures

1.1 Reference architectural framework for proprioceptive nodes 7
1.2 The self-aware node: conceptional component diagram 8
1.3 The self-expressive node: conceptual component diagram 9

2.1 View of the SystemC model of the proprioceptive node 13
2.2 Interconnection mechanism between proprioceptive nodes 19
2.3 View of the SystemC model of a proprioceptive node 20
2.4 View of process cycles 21
2.5 Example of transaction groups A and B are processes 22
2.6 Transaction groups for the processes A1 (left) and A2 (right) WRITE indicates that the transactions initiated here are write transactions 24
2.7 Transaction group for processes B - WRITE indicates that the initiated transactions are write transactions 24
2.8 Transaction groups for processes C1 (left) and C2 (right) WRITE and READ indicate the type of transactions that are generated. WRITE for write transactions, READ for read transactions 25
2.9 Transaction group of process D 26
2.10 Transaction groups for processes E1 (left) and E2 (right) READ indicates that the generated transactions are read transactions 27
2.11 Prozess F 28
2.12 Implemented chronology of processes 30
2.13 Delta cycles between the synchronization points in a node 36
2.14 Transactions flows for the processes A1, A2, B in each process cycle . . . 39
2.15 Transaction flows for the processes C1, C2, E1, E2 and D per process cycle 40

List of Tables

2.1 TLM-components, role and number 16
2.2 TLM-components und sockets 17
2.3 components, ports and channels 19

List of abbreviations

A

Abbr.	Abbreviation

E

EPiCS	Engineering Proprioception in Computing Systems

F

Fig.	Figure

R

RTL	Register Transfer Level

T

TLFT	Thread-Level Fault Tolerance
TLM	Transaction-Level-Modelling

1 Proprioceptive computing nodes

[?] This chapter gives the definition of proprioceptive nodes and describes their reference architectural framework developed within the EPiCS project and on which basis the simulation model is later developed.

1.1 Definition and properties of proprioceptive computing nodes

A computing node is referred to as *proprioceptive*, if it exhibits the following properties: Self-Awareness, Meta-Self-Awareness and Self-Expression.

A node is self-aware if it possesses information about its internal state on the one hand, and on the other hand, sufficient knowledge of its environment (including other potential nodes) to determine how it is perceived by the other parts of the wider system. If a node reacts as a result of information about its internal state, then this is called *private* Self-Awareness. But it reacts as a result of information about its environment, it called *public* Self-Awareness.
A node is self-expressive if it is able to assert its behaviour upon either itself or other nodes. This behaviour is based upon the node's state, context, goals, values, objectives and constraints.
Finally, Meta-Self-Awareness represents the highest level of self-awareness that a node can exhibit. It enables logical reasoning to the node about its own ability to be self-aware.

These properties are not intended to replace or supersede other self-* properties of a system. Self-Awareness and Self-Expression in nodes of a system should rather be considered as part of the engineering process of a system and as those basic properties which help achieve other self-* properties such as self-adaptation or self-organisation.

Proprioceptive nodes are conceptual components or sub-systems, i.e. they are not required to physically exist as separated components within an application but provide a logical structure for the reasoning about interactions between parts of a system, where these parts can have different levels of knowledge, autonomy and distributed decision making. In this sense, a node can be thought of as an agent. A collective of proprioceptive nodes is called a proprioceptive computer system.

1.2 Reference architecture of a proprioceptive node

The reference architectural framework for proprioceptive nodes is shown in the figure 1.1. It consists of many conceptual components and can be divided into three main parts based upon the definition of the proprioceptive node. These are: the self-aware node, the self-expressive node and the monitor or controller (responsible for the Meta-Self-Awareness). The arrows in the figure shows the information flow between the individual components.

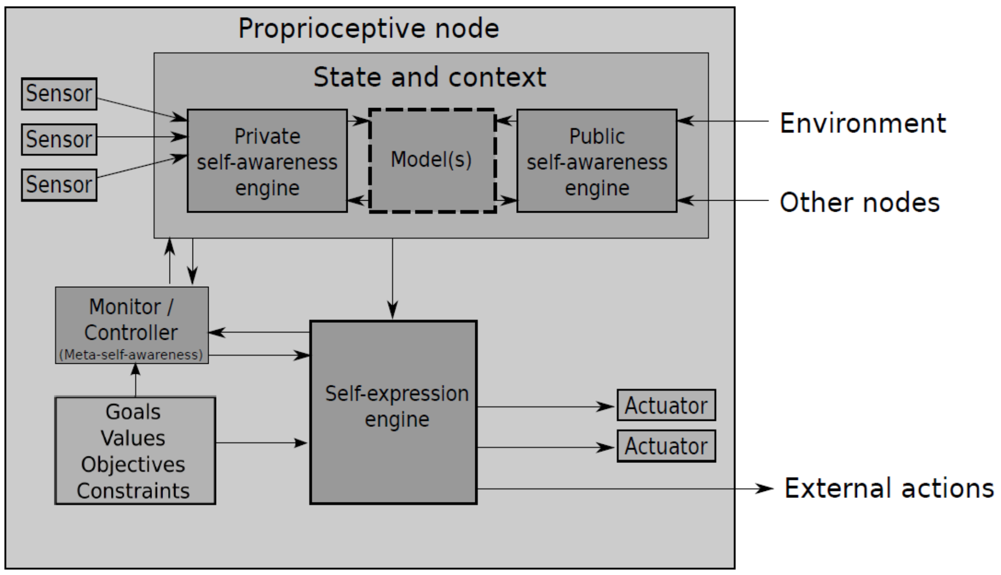


Figure 1.1: *Reference architectural framework for proprioceptive nodes.*

1.2.1 Self-aware node

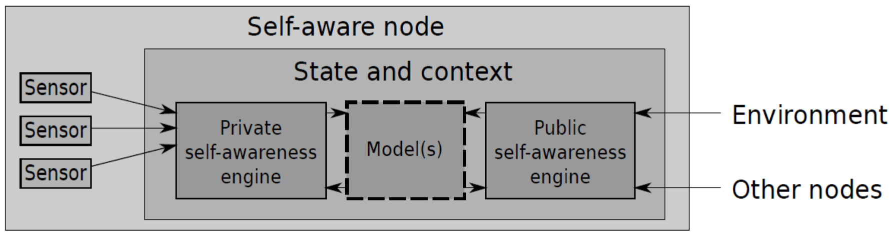


Figure 1.2: *The self-aware node: conceptional component diagram.*

The figure 1.2 above illustrated the self-aware node. It consists of the following components:

1.2.1.1 Sensors

In agent architectures information is gathered through sensors. This is why the reference architectural framework for proprioceptive nodes comprises sensors. The are two different types of sensors: internal and external sensors. The internal sensors gather information about the node internal state, by which means self-awareness is achieved. This type of sensor can gather information, for example, concerning the actual battery charge level, the internal temperature or the CPU load. External sensors, which are illustrated by the Environment and Other nodes components in the figure 1.2 above, gather information from external sources of the node. Examples of this can be a sound level detector, a camera providing images from the environment or a receptor from messages from other nodes. By using external sensors, a node is able to control his environment including importantly any effects that it has upon it.

1.2.1.2 Private and public self-awareness engines

These engines collect and process information from each sensor type of the architecture. The private self-aware engine collect and process information from the internal sensors while the public self-aware component collect and process information from the external sensors. Both takes decision about whether in view of the data processing results actions must be taken. For example, the private self-aware component monitoring the available hard disk space might collect information regularly from the disc space sensor and simply discard it as long as the value doesn't drop below a predetermined lower bound. But in case that the free space exceed that bound, the node may need to take an action. The self-aware engines are responsible for this initial processing.

1.2.1.3 Model(s)

In addition to purely instantaneous sensor information, both private and public self-aware components might possess historical knowledge or knowledge about the likely effect of potential future actions or decisions or contextual information. To enable this richer form of self-awareness, the self-aware engine may engage in learning or modelling information. This is why the architecture of a self-aware node introduce the possibility of optional internal models (dashed box in figure 1.2). These models can be included, when necessary, to achieve the required higher level of self-awareness in the node. The self-aware components may therefore either be minimal, simple processing units or scaled up to engage in any range of machine learning or decision-making techniques, as required. This scalability is a key feature in this architecture, supporting both minimal and highly advanced self-aware capabilities

Furthermore, the self-aware engines may be adaptive and self-learning, and themselves learn improved behaviour during runtine. Thus, their behaviour don't need to be fully specified in advance but instead may be learnt, for example through the maximisation of an utility function. In this way, the processing of sensor information is a task like any other one that the agent should perform, and agent techniques may be applied to this task.

1.2.2 Self-expressive node

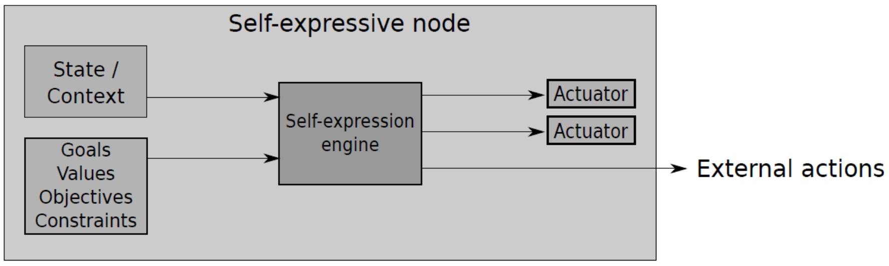


Figure 1.3: *The self-expressive node: conceptual component diagram.*

The picture above shows the self expressive node. It consists of the following conceptual components:

1.2.2.1 Actuators

The first key property of a self-expressive node is that it must able to assert its behaviour upon itself or other nodes. To achieved this, many agent architectures make use of actuators. Hence, the architecture of the self-expressive node defines actuators. Based on the expected characteristic of the self-expressive node mentioned above, they are two type of actuators defined here: the internal actuators, which are capable of carrying out actions which make changes to the internals of the node, and external actuators, which are capable of carrying out actions which have effect outside the node. So, actuators represent the actions available to a node. The distinction between internal and external actions depend clearly on the node boundaries within the whole system. Example for internal actions could be the throttle of a CPU fan (where the CPU and its heat management are parts of the node), the re-compilation of some software with different optimizations (where such software are part of the node's function). Internal actions can be observed from the outside of the node and from other nodes of the whole system. External actions in contrast can be observed from the outside of the node and from other nodes and are more likely to be designed to be so. Example of external actions could be the sending of messages to other nodes, the modification of the environment around the node, the relocating of the node functionality within the whole system.

1.2.2.2 Self-expressive component

Before an actuator perform any actions, there are some decisions to make beforehand according to the node's strategy and behaviour about which actions to take and when the should be performed. The self-expressive engine is the component responsible for that decision-making in the node's architecture. It embodies the node's strategy in the architecture and thereby has the sole control over the node actions.

1.2.2.3 Goals-Values-Objectives-Constraints component

The second key property of a self-expressive node is that the node's behaviour must always based upon its state, context, goals, objectives and constraints (in abbr. g.v.o.c.). The goals, values, objectives and constraints are available within the node through this component and are used by the self-expressive component to determine the actions which should be carried out to assert the node's behaviour, as seen before. The state and contexts relate to the node's knowledge about itself and its environment. The goals, values, objectives and constraints in contrast relate to the desired behaviour of the node.

- **Goals** are high-level description of desired outcomes or states. They may be required to be obtained only once or permanently (ongoing desired state). Goals may conflict or otherwise be in tension with other goals.
- **Objectives** are specific descriptions of measurable outcomes which support the achievement of the goals. A goal can therefore be validated by the achievement of each of its supportive objectives
- **Values** provide the knowledge required to express preferences between goals which are in tension. They may also be used to decide between alternative actions in advancement of conflicting actions, where a trade-off exist between two or more goals.
- **Constraints** are limits within which the node must operate, which may be imposed physically or by design. The violation of a constrain indicates that the node has performed improperly or attempted an illegal action. The available actions must be determined such that they do not violate constraints.

1.2.3 Monitor/Controller

The monitor is the conceptual component responsible for the meta-self-awareness of the node. So, its task consists in monitoring the behaviours of the self-expressive and self-aware engines and, when necessary, modifying this behaviour in order to make them better meet the node needs. To be able to do this, the monitor has access to the node's knowledge about its internal state and context (so to the self-aware node). It also has access to the knowledge of the self-expressive component and, lastly to the goals, values, objectives and constraints of the node. This is illustrated by arrows in the figure 1.1. In summary, the monitor has a high-level view of the node's behaviour and can intervene, as and when required, to direct or refocus lower level of self-awareness and self-expression behaviour in the node.

2 Modelling of the proprioceptive computing node

As already mentioned, the first goal of this work, is to develop a model for proprioceptive computing nodes which can be used for test purposes within the EPiCS Project. The developed model shall be extensible, configurable and implemented in SystemC based on the reference architectural Framework described in the previous chapter. The following sections show the proceedings as well as the used SystemC concepts for modelling the model said.

2.1 Preliminary decisions

SystemC models can either be created at the register-transfer level or at a higher lever, the transaction level which is in fact the core area of SystemC. Therefore, the first step by the modelling here consists of identifying the appropriate abstraction level for this model.

At the register-transfer level, signals and port connections of a system are cycle-accurate. This results in very long simulation times. The transaction level modelling (abbr. TLM), in contrast, achieves a substantially higher simulation performance because of the following two main reasons: the focus by modelling is on the transactions between the components and the time behaviour is strongly abstracted. The transaction-level modelling is thus appropriated for the analysis of a system framework. In view of the above statements, we decided to model the proprioceptive node at the transaction level.

The choice of the transaction level modelling of the proprioceptive node raises the question of which coding style shall be used for the time sequences in our model. For this purpose the library of SystemC TLM offers two options: the loosely-timed (abbr. LT) and approximately-timed (abbr. AT) coding styles. To answer that question, the knowledge, on the one hand, that the focus shall be on the simulation performance was of importance and on the other hand, the knowledge that the project do not attached any importance to the time sequences details of the individual transactions between the components of the defined system. Those two facts led to the choice of the loosely-timed coding style, because the highest simulation speed can be achieved with this coding style and the temporal processes are modelled only roughly.

2.2 Identification of the TLM-components and their roles

After the question concerning the level of abstraction and of the emerging issue regarding the coding style to be used for the time sequences were clarified, the next step here is to identify the TLM components of which the model will consist as well as their respective roles in the model.

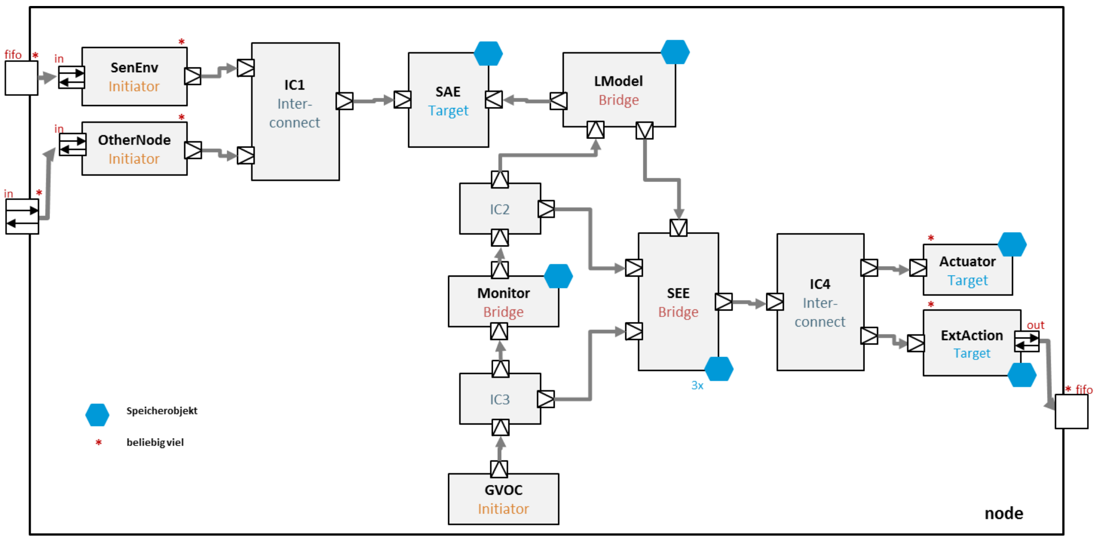


Figure 2.1: *View of the SystemC model of the proprioceptive node.*

Using the given definition of the different roles of TLM components and based on the given description of the developed architectural framework for proprioceptive nodes, we were able to identify or rather determine the following TLM components for our model (see figure 2.1):

- SenvEnv: This TLM component represents the internal and the external sensor (Environment) of the node. A node can contain at least one SenvEnv and as many as required. The necessary number must be specified by the user before simulation start and is generated during elaboration using those specifications. Each generated sensor will access a particular type of information and initiate write transactions to pass this information to the TLM component SAE, which is described below. Consequently, the SenvEnv-TLM-component is an initiator.
- OtherNodes: The TLM Component OtherNodes is also an initiator in our model and represents the external sensors which access information from other nodes in a proprioceptive computing system. A node can contain as many OtherNodes as required. Here also, the necessary number must be specified by the user and the components are generated during elaboration using the user specifications.

- SAE: This TLM component will collect or rather store the information sent by all the internal and external sensors presented above. It is therefore a target.

- GVOC: This TLM component, as the name suggests, embodies the Goals-Values-Objectives-Constraints component of the proprioceptive node. So, it will provide the goals, values, objectives and constraints of the proprioceptive node to the monitor and the self-expressive engine. In order to do that, it will initiate write transactions. The GVOC component is therefore an initiator.

- LModel: This component represents the part of the self-aware component which is supposed to process the received sensor information. Because these are stored in the TLM component SAE, it will first initiate read transactions to readout each piece of information from the SAE memory and then evaluate it. If the evaluation results indicate that actions has to be taken, this component will then inform the self-expressive engine of the node - represented here by the TLM component SEE below - using write transactions. Furthermore the LModel will document his actions on his so-called report memory.

- SEE: As already mentioned above, this TLM component embodies the self-expressive engine of the proprioceptive node. Thus, it will take decisions about what actions to take according to the received data from the self-aware engine, here the TLM component LModel, and with respect to the g.v.o.c. data. It will then communicate his decisions to the corresponding actuators, which will then perform the selected actions. The choice of the appropriate actuators as well as the data transmission are going to be performed by write transactions. This implies that this component is an initiator. It is also a target for the TLM components GVOC and LModel, as seen above. Furthermore the SEE component will document his actions on his own report-memory, like the TLM-component LModel.

- MONITOR: This component represents the monitor of the proprioceptive node. As described in the previous chapter, the main task of the monitor is to control the handling of the self-expressive and self-awareness engines. To this end, this TLM component will read the report memory spaces of both his targets in user-defined successive time intervals and process the data. This component is therefore an initiator. Further, the fact that it receives data from the TLM component GVOC indicates that it is also a target. The monitor or controller of the proprioceptive node also has the capacity to intervene in the node, when required, to redirect or refocus lower level of self-awareness an self-expression in the node, as mentioned in the previous chapter. This functionality wasn't yet required and is therefore not implemented in this TLM component.

- ACTUATOR: This represents an internal actuator of the node. It is the target component of the TLM component SEE, so the self-expressive engine. As the proprioceptive node may have many actuators, the user will also have the possibility to define the number of actuators to be generated in the model.

- ExtAction: This represents an external actuator of a proprioceptive node and is, like the actuator component, the target of the self-expressive engine during transactions. Such as for the actuator described above, the number of ExtAction in a node will be specified by the user before simulation start.
- ic1, ic2, ic3, ic4: These are the interconnect components of the model. They are used where several initiators communicate with the same target or an initiator has to communicate with several targets over the same transaction type, . They will forward each method call to the corresponding target and carry out address coding and encoding, when necessarily and before forwarding the calls. ic1 is located between the initiator components SenvEnv and OtherNodes and their target component sae. ic2 is the interconnect component between The monitor and its targets. ic3 routes the method calls of the gvoc component to its targets. ic4, lastly, is located between the TLM component see and its targets, the actuators of the node.

Additionally to all the TLM components stated below, there is also a SystemC module which represent the next and also the highest level in the model hierarchy. This component is termed node and is responsible for the generation of all the TLM components in a node with respect to the user specifications and also for the ports and sockets bindings that are indispensable for the simulation. Further, it contains a process which contributes to the supervision or rather regulation of the process executions in a node. (see 2.4.3.6).

Table 2.1 illustrates the TLM components of the model in a summarized form.

Table 2.1: *TLM-components, role and number.*

Description	**Task**	**Role**	**Number**
SenEnv	sends internal and external (Environment) sensor data to SAE	Initiator	≥ 1
OtherNode	sends external (from other nodes) data to SAE	Initiator	≥ 0
SAE	acts as a storage for all sensor data	Target	$= 1$
GVOC	sends the goals, values, objectives and constraints data to the SEE and the Monitor	Initiator	$= 1$
LMODEL	processes the sensor data	Bridge	$= 1$
SEE	Decision maker for the node strategy; receives information of LModel and activates actuators	Bridge	$= 1$
MONITOR	monitors the SEE and LMODEL	Bridge	$= 1$
ACTUATOR	embodies the internal actuator	Target	≥ 1
ExtAction	embodies the external actuator	Target	≥ 0

2.3 Modelling of the communication systems

To model the communication systems or mechanisms between components at the transaction level, the TLM Library of SystemC provides sockets and a set of six interfaces which we are going to make use of here.

From the task description of the TLM components above, it appears that they may communicate between each other not only inside but also beyond the node boundaries (e.g.: to pass data to other nodes). Concerning the communication between proprioceptive nodes, they are no specific requirements made. So, for this communication part (communication with the outside world), we decide to use ports and channels to model an asynchronous communication between nodes, which in no means run contrary to the definition and the functionality of a proprioceptive system.

SystemC and his TLM library define different types of ports, channels and sockets which provide different mechanisms of connection and application areas. Concerning the interfaces, a distinction is also made between blocking and non-blocking transport interfaces whose respective application area depends on the modelling styles used for the time sequences of the transactions

The connection elements used here are presented in the following sections and along with it, the reasons for their selection.

2.3.1 Interfaces and Sockets

To achieve a high simulation performance, we decided to use the loosely-timed coding style for the time sequences of the transactions in the model (see section 2.1) . This involves the use of the blocking transport interface. So, all the TLM components of a node will communicate with each other exclusively on the forward path by calling the blocking transport method b_transport, which belongs to that interface.

The interface method calls occur via sockets. Hence, each TLM-component must have at least one socket to be able to communicate with others. The type of the socket depends on the role of the component; an initiator component must have an initiator socket, a target component must have a target socket and an interconnect component must have at least one of each. Further, for maximal interoperability, it is recommend to use standard sockets instead of the convenience ones. For this reason, the sockets used in this model are mostly standard sockets, as presented in table 2.2. The convenience

Table 2.2: *TLM-components und sockets.*

Description	**Role**	**Initiator Socket**	**Target Socket**
SenEnv	Initiator	Standard	-
OtherNode	Initiator	Standard	-
gvoc	Initiator	Initiator	-
sae	Target	-	Standard
actuator	Target	-	Standard
ExtAction	Target	-	Standard
lmodel	Bridge	Standard	Standard
see	Bridge	Standard	Simple Tagged
monitor	Bridge	Standard	Standard
ic1	Interconnect	Standard	Simple Tagged
ic2	Interconnect	Standard	Standard
ic3	Interconnect	Standard	Standard
ic4	Interconnect	Standard	Standard

sockets used here, precisely the tagged simple target sockets, are only for the interconnect component ic1 and the bridge component see. ic1 comprises a vector of the stated convenience target sockets, whose size corresponds with the number of required sensors (SenvEnv and OtherNode) specified. In the TLM component see, there are three of these sockets defined.

Based on the given definition of the tagged simple sockets, we know that each interface method call, which arrive in a target component through these sockets, are tagged with a self-determined *id*. In that target component, this id allows the identification of the socket through which the call arrived and thus, also with the help of the bindings, the identification of the component which initiated the call. In the interconnect component IC1, the id is then used to map the given start address of the transaction to the corresponding address in the SAE memory space and, thereby ensure that each sensor always access its allocated memory space and also to avoid data overwriting (see section 2.4.5.3). In the TLM component SEE, the id is rather used to choose the right storage object for incoming data during transactions.

As for the bus widths, which are modelled by the template parameter BUSWIDTH of the sockets, the user has the possibility to specify them (a total of five) during the node instantiations. Each value must be a positive integer and a multiple of 8 and will substitute the default value (32) of the template parameter BUSWIDTH.

2.3.2 Ports and channels

As already mentioned in the introduction of this chapter, there is not only an intercommunication between TLM components within a node but TLM components should also be able to communicate with other TLM components of other existing nodes. This concerns the TLM components SenEnv, OtherNode and ExtAction.

Each TLM component ExtAction should be able to send data to the outside of a node. Hence, each has a output port of type SC_FIFO_OUT_IF which gives him write access to a fifo channel. On the other side of the communication, that is the TLM component OtherNode in another node, there is a input port of type SC_FIFO_IN_IF with which the linked OtherNode component can read out the data available on the same fifo channel. The fifo channel in question, is defined in the same component NODE as the actuator ExtAction.
The following illustration 2.2 uses the example of a proprioceptive system which consists of two nodes proprioceptive nodes to show the communication mechanism and the elements used for the node intercommunication in this model, as described above. For reasons of space, only the elements taking part to that communication are shown.

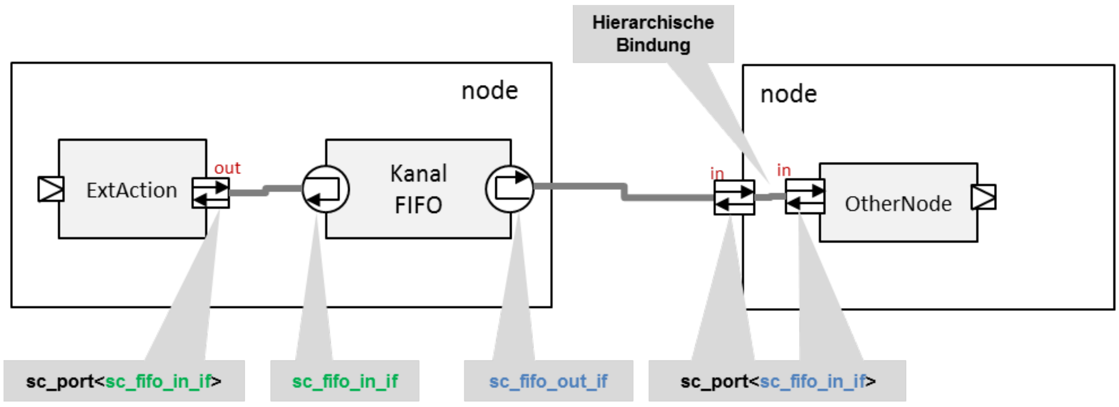


Figure 2.2: *Interconnection mechanism between proprioceptive nodes.*

As for the sensor data, they are initially in binary or text files. In order to transfer them into the corresponding sensor components at simulation start, we also make use of ports and fifo channels. Each TLM-component SenEnv has an input port of type sc_fifo_in_if, so with write access to a fifo channel. For each existing TLM-component SenEnv, there is also a fifo channel to which the input port just mentioned is bound to. In addition, there is a method process whose task is to read the files and to write the data on the corresponding fifo(s) always before the processes of the sensor components run. The sensors then read out the fifo channel in each process-cycle via their ports and can then transfer it to the SAE component.

Table 2.3: *components, ports and channels.*

Description	**Port**	**Channel**	**Number**	
SenEnv	input port	-	$= 1$	
ExtAction	output port	-	$= 1$	
node	input ports	FIFO	≥ 1	≥ 1

The TLM Model of the proprioceptive node with his TLM components and the connecting elements presented above are shown in the following figure; here, the arrows represent the transactions direction or rather the data flows. For channels, the representation of the interface has been omitted owing to space constraints.

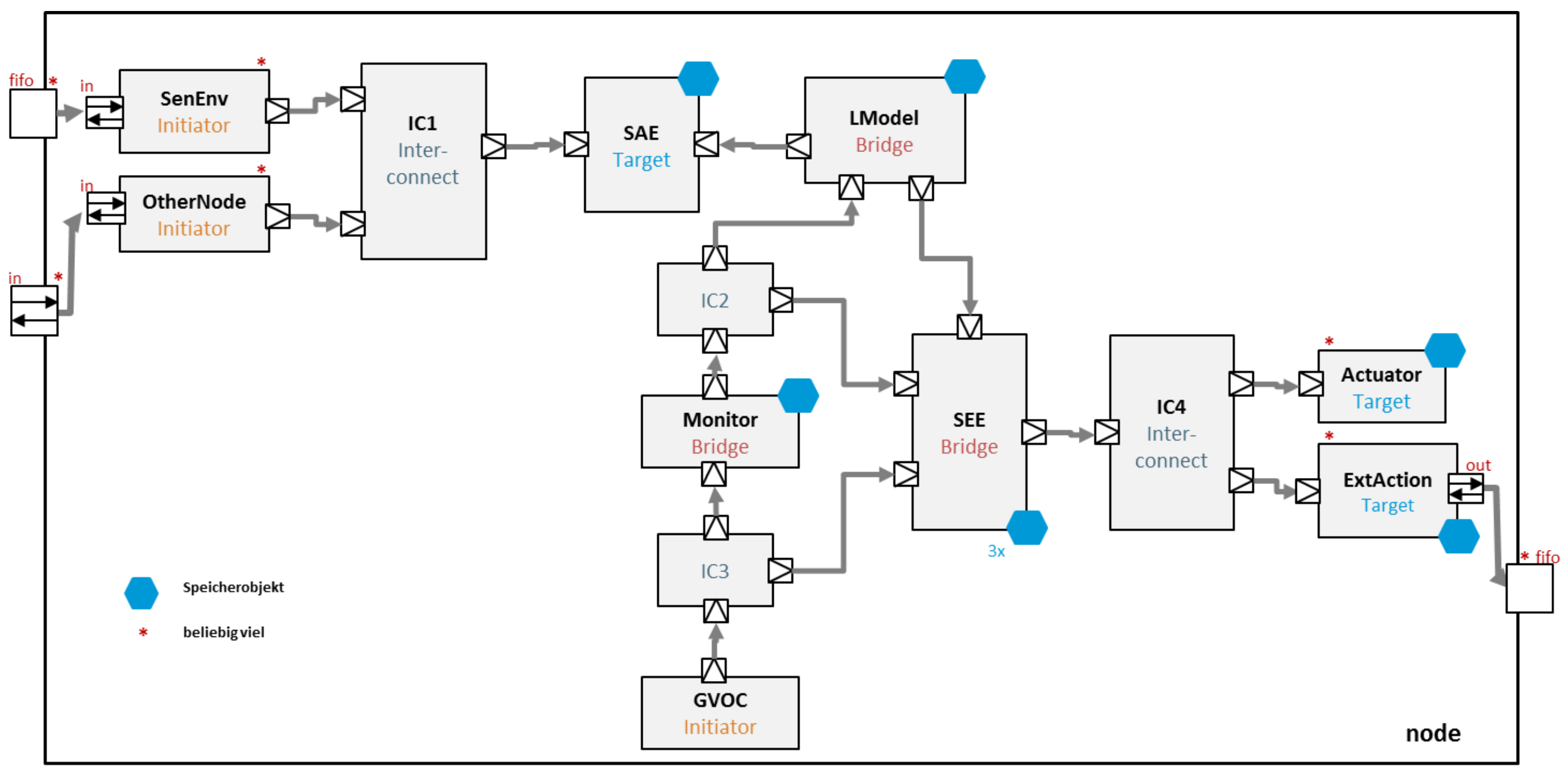


Figure 2.3: *View of the SystemC model of a proprioceptive node.*

2.4 Processes and transactions

After the components identification and that of the connection elements and mechanisms, describing the behaviour of each of the stated component is the next step.
To describe the behaviour of a SystemC module at the register-transfer level as well as the behaviour of a TLM component at the transaction level, processes are used. There are three different type of processes: method, thread and sc-thread processes. Since we chose the transaction level modelling and the loosely-timed coding style to model the temporal sequences, there is no other possibility than to use thread processes to describe the transactions between the TLM components and these thread processes must be temporally decoupled.

The following sections present the existing processes as well as the implementation of the temporal decoupling of these processes in the model. In addition, the execution chronology of the processes between the synchronization points set by the global quantum will be shown. The reasons for that chosen chronology and the mechanisms used to always ensure it will then be presented. At last, the transaction sequences in the model will also be illustrated with the help of sequence diagrams. But , firstly, two terms which were created within the scope of this work and which play an important role for the understanding of the following sections will be defined.

2.4.1 Definition of some important terms

Process cycle

As *process cycle*, we indicate the period of time, within which all processes from all nodes of a simulated system are executed, after a synchronization point up to the next synchronization point. This period is between the synchronization points set by the global quantum. In each process cycle, every process synchronizes only once.

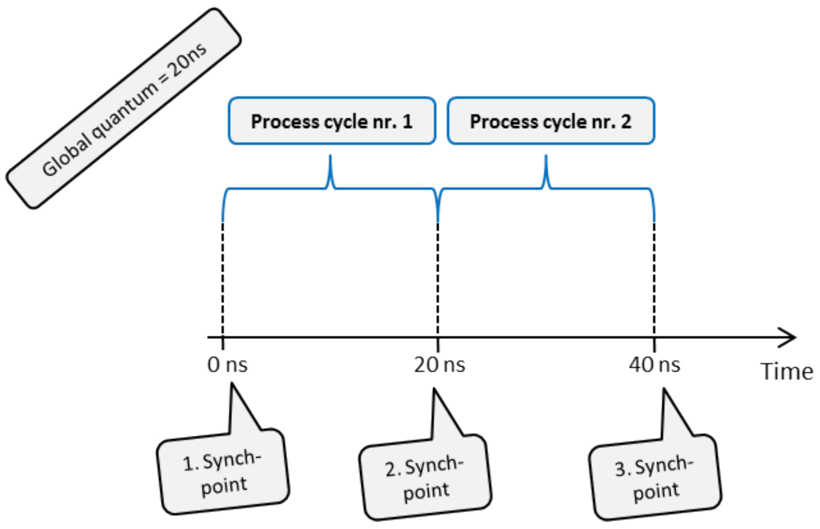


Figure 2.4: View of process cycles.

Transaction group

As *transaction group*, we indicate the TLM components initiator, target and/or interconnect, which always communicate with each other with the help of one and the same process and by executing the same type of transactions, either write transactions or read transactions.

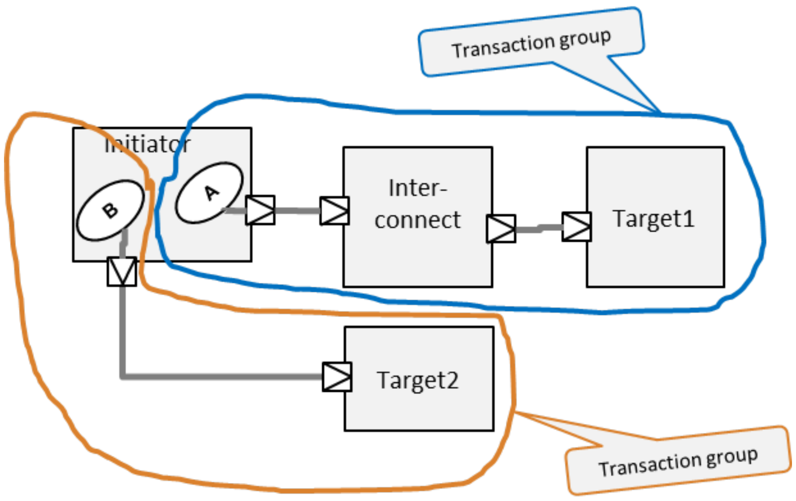


Figure 2.5: Example of transaction groups
A and B are processes.

2.4.2 Temporal decoupling

As already mentioned, processes must be temporal decoupled in this model. Thus, there is a quantum keeper defined in each of the TLM components of the node model and also in the module NODE.

The first step to implement the temporal decoupling is to set the global quantum and recalculate the local quantum by respectively calling the method set_global_quantum() with a time argument and the method reset(). This time value is given by the user while instantiating the nodes of the proprioceptive system to be simulated and applies for all processes of the system.

The next step is to calculate the local time offset in each process during the simulation. For that purpose, we define the variable t_{sum_delays} in each process as a time object which is then used as the required time argument for every call of the blocking transport method $b_transport$. The latency of every transaction t_{trans_delay} is then be added to that variable by the target component. So:

$$t_{sum\_delays} = t_{sum\_delays} + t_{trans\_delay}$$

This latency t_{trans_delay} is calculated by multiplying the burst length BL and the latency of a single transaction,

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

The latency of a single transaction, in its turn, is calculated automatically during elaboration with the help of some user specifications (see 2.4.5.5). After each transaction, the method set() is finally called with the updated time variable t_{trans_delay} as argument.

Thereupon follows the evaluation of the synchronization condition and the call of the synchronization method sync, if that condition is fulfilled; this means that the process has reached the next synchronization point and now suspended until the simulation time reaches its time. Otherwise, the next transaction is started.

2.4.3 Behaviour descriptions

In Addition to the operations presented in the previous section for the implementation of the temporal decoupling, processes also have to generate transactions. These transactions are described in the following sections. For each process, a figure shows the transaction group which is concerned and the component of this transaction group in which that process is located. Furthermore, the transactions generated by the process are described.

2.4.3.1 Processes A1 and A2

Processes A1 and A2 are located in the initiator component GVOC, as shown in figure 2.6. They transfer the specified goals, values, objectives and constraints into the node by calling the blocking transport method (b_transport). Before every such call, the attributes of the transaction object passed as argument are set. There is a transaction object for each of both processes. Process A1 transfer the stated data to the MONITOR and process A2 to the component SEE, so the self-expressive engine of the node.

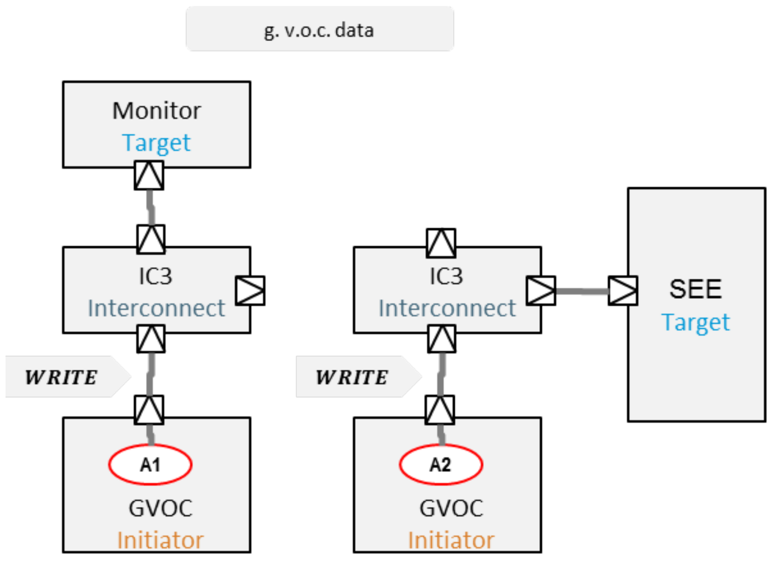


Figure 2.6: *Transaction groups for the processes A1 (left) and A2 (right) WRITE indicates that the transactions initiated here are write transactions.*

Because of the important role that play the g.v.o.c. data in the node, processes A1 and A2 are always the first processes to be run in each process cycle and they always execute without any suspension till their respective next synchronization points.

2.4.3.2 Processes B

The processes B represent all the processes located in the sensor components of the node, i.e. the TLM components SenEnv and OtherNode. Each of them has a process B which calls the blocking transport method `b_transport` to transfer its data to the TLM component SAE through the interconnect component IC1. This is graphically presented in the following figure.

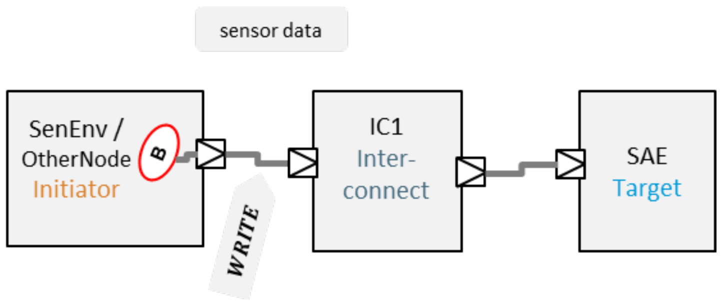


Figure 2.7: *Transaction group for processes B - WRITE indicates that the initiated transactions are write transactions.*

Before every method call, each sensor component set the attributes of its transaction object passed as argument. The memory space owned by the TLM component SAE is equally shared by the sensors in each node. In other words, each sensor has its own memory area on the data storage object owned by the TLM component SAE. Every sensor knows only the size of its memory area, but not the address range. So the value of the address attribute will always be adjusted by the interconnect component IC1 for each incoming method call and before the call is forwarded (see section 2.4.5.3).
Given the fact that the processes B generates transactions to transfer sensor data to the self-awareness engine of the node, they will always be run next after the processes A1 and A2 in every node and every process cycle, and also ahead simulation time until they reach their next synchronization points, as is the case for the processes A1 and A2.

2.4.3.3 Processes C1 and C2

These thread processes are located in the TLM component LModel.
Process C1 is responsible for the linear readout of the memory space of the TLM component SAE . By its every execution, process C1 read out one dataset whose length is specified by the user before the simulation start. This dataset is then forwarded into the module for processing. After the data processing, the TLM component LModel decides whether some measures has to be taken in view of the processing results. Its decision is finally sent to the TLM component SEE by the process C2.

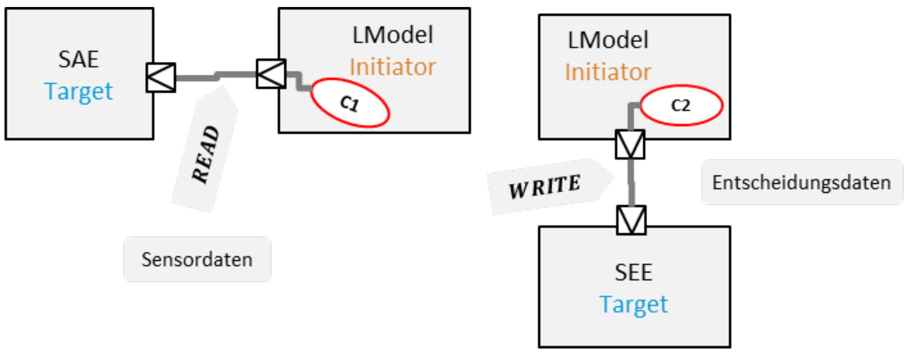


Figure 2.8: *Transaction groups for processes C1 (left) and C2 (right)*
WRITE and READ indicate the type of transactions that are generated. WRITE for write transactions, READ for read transactions.

To achieve the aforesaid, process C1 will always be the first of both C1 and C2 to be executed in each node and in each process cycle next after all the processes B were suspended. By every execution, process C1 creates the following notifications successively after a transaction and the decision-making mentioned above: an immediate notification and an delta notification (wait(SC_ZERO_TIME)). With the delta notification, the process C1 is suspended and with the immediate notification, an event belonging to the dynamic sensitivity of process C2 is notified. Process C2 therefore run immediately after C1 is suspended. By every execution, process C2 generates a transaction to transfer the decision data of the TLM component LModel to the self-expressive component . To inform the self-expressive component, process C2 executes an immediate notification after that transaction is completed. The immediate notification is created by calling the function notify without any time argument. The so notified event belongs to the dynamic sensitivity of process D. Finally, process C2 executes a timed notification by calling the function wait with SC_ZERO_TIME as time argument. It's therefore suspended and process D is the next on the set of the runnable processes.

In addition to the notifications cited above, process C1 first always executes a delta notification. This delta notification is there to ensure that process C2 never missed the immediate notification of its events performed by C1. This is more detailed in the section 2.4.4.1 below.

2.4.3.4 Process D

This is the thread process describing the behaviour of the TLM component SEE. Thus, it is run after process C2. By every execution, it first readouts the decision data received from the component LModel and evaluates it. Depending on these data, the TLM component SEE is able to identify whether the component LModel had detected a problem while evaluating the sensor data or not. After the data evaluation, the TLM component SEE decides on the measures which must be taken by the node. And after the decision-making, process D initiates write transactions to the corresponding actuators, ExtAction- or Actuator components, which then execute the chosen measures and use the response status attribute of the passed transaction object to inform the TLM component SEE as to whether they could successfully perform the measures or not.

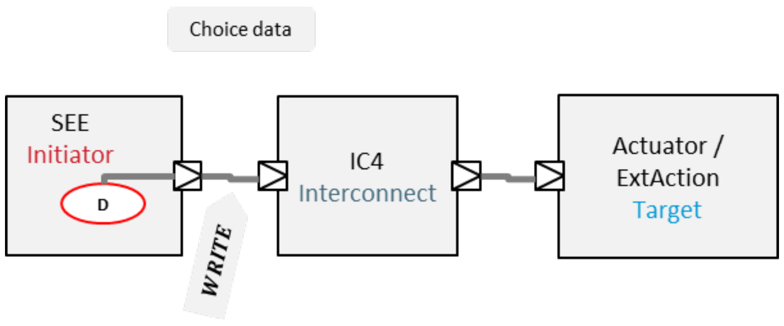


Figure 2.9: *Transaction group of process D.*

2.4.3.5 Processes E1 and E2

These are the threads processes describing the behaviour of the TLM component Monitor, more precisely its monitoring task on the TLM components LModel and SEE. Process E1, which is responsible for LModel, is run in every process cycle and in each node immediately after process C1 and process E2 immediately after process D. By its every execution, process E1 readouts the report memory of his target and E2 the report memory of his target component SEE.

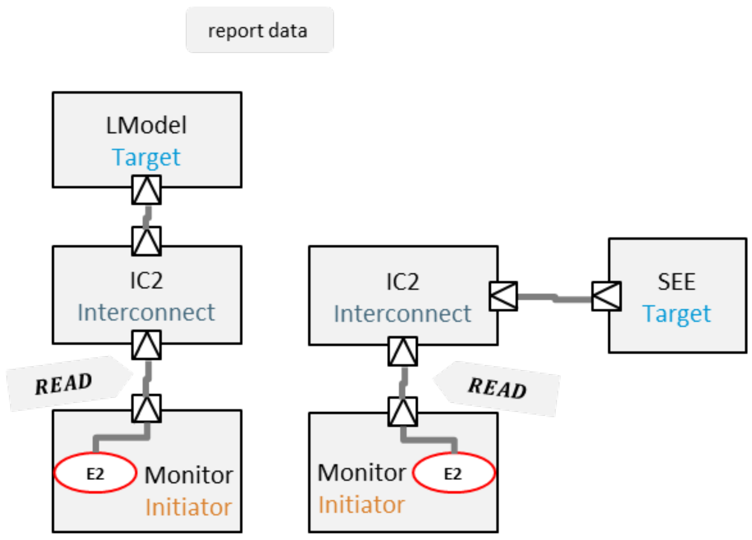


Figure 2.10: *Transaction groups for processes E1 (left) and E2 (right)*
READ indicates that the generated transactions are read transactions.

In view of the fact that the monitor of the proprioceptive node doesn't have to constantly monitor the self-expressive and the self-awareness engines, these processes will only be running at regular intervals of process cycles. The interval between their executions must be specified by the user (as a positive integer value) for each node and before the simulation start. During the simulation, a counter is increased by one at the beginning of each process cycle and the updated value is then compared with the user specification. In case that the counter value is equal to the specified number, events belonging to the dynamic sensitivity of processes E1 and E2 are notified (immediate notifications) and the counter is reset. The immediate notification, which activates process E1, is executed by process C1 and the one which activates process E2 in D.

In each process cycle, E1 and E2 are so often executed as the processes C2 and D once they are activated. By each execution of E1 and after the generated read transaction mentioned above is completed, an event belonging to the dynamic sensitivity of process C2 is notified by calling the function notify() without time argument. Immediately after that immediate notification, Process E2 is suspended by a call to the function wait() with SC_ZERO_TIME as time argument. Thus, process C2 is executed immediately after it. Unlike process E1, process E2 only executes a delta notification after the read transaction is completed.

2.4.3.6 Synchronization process F

This thread process is located in module NODE (see figure 2.11), which is, as already mentioned, the highest level in the module hierarchy.

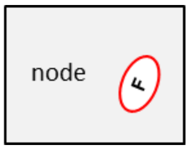


Figure 2.11: *Prozess F.*

Process F is run in each process cycle only once and synchronizes always immediately after it has started. So, there is no call of the blocking transport method, thus no transactions generated. The purpose of this process is to ensure that in all nodes of a simulated multi-node proprioceptive system the process cycles end at the same time. 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 respective synchronization points. The next process cycles then start after the kernel has advanced the time till the next quantum.

Without process F, the control will be yielded back to the kernel immediately after all processes of a node have reached the next respective synchronization point. The simulation time will consequently be advanced although some processes of other system nodes haven't reached their next synchronization points. These processes will then continue their execution in the following process cycle and at the same simulation time as the ones which got back from suspension and eventually in the same delta cycles. Knowing that the execution order of processes within a delta cycle is not implementation-defined, this could lead to data inconsistencies, if the running processes initiate read and write accesses to a shared memory space.

2.4.4 Execution chronology

From the previous behaviour descriptions of the respective processes, it arises that processes of a node always run in the same chronology within a process cycle, which is as follows:

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

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

The following sections show how this indispensable chronology of processes is implemented. This is shown graphically beforehand in figure 2.12.

2.4.4.1 Processes A1, A2, B's and C1

Because the order in which the runnable processes are executed within a delta cycle is not implementation-defined, processes with different access rights on a shared memory area should not run in the same delta cycle. Indeed, this would lead to data inconsistencies. To avoid it, processes B and C1 always start running at different times in each process cycle, as illustrated in the figure 2.12.

The figure 2.12 above also shows that the execution start times of processes A1 and A2 also differ, although they don't even access the same memory areas. Furthermore, processes B of the TLM components SenEnv and the processes B of the TLM components OtherNode, which both have read access to their target memory area, have different execution start times. The purposes here is just to have a better overview of the transactions in the output files and make their tracking much easier.

To set these execution start times, we used two elements which, in addition to the global quantum, also play a decisive role in the determination of the model times to be simulated when using temporal decoupling. These are:

- the timed notifications and
- the local time offset (preceding a process synchronization)

Timed notifications
As already known, a timed notification results from a call of the function wait() with a non-zero-valued timed argument. In each of the processes A2, B's, C1 and F of this model, there is a wait()-statement with a non-zero-valued timed argument (t_{beg}) placed before the *while*-loop. So, this function call is only executed once after simulation start, namely in the first evaluation phase after the initialization phase. Furthermore, the time argument passed is different for each of the stated processes. In doing so, we create timed notifications which postpone the first transaction execution in each process instance to the passed time argument.
Assume for example, that the time unit in the model is *ms* and that the passed time argument for process C is 0.4 *ms* . This means that C executes it first transaction at 0.4 ms after the simulation start.
As for process A1, there is no wait() statement. So, it is always the first process that is run in each process cycle and it first transaction is at time *zero* of the simulation.

Local time offset
A temporally decoupled process run ahead simulation time. A call of the member function sync() of the quantum keeper suspends a temporally decoupled process for a period of time, namely the local time offset t_{off}. The suspended process can run again only

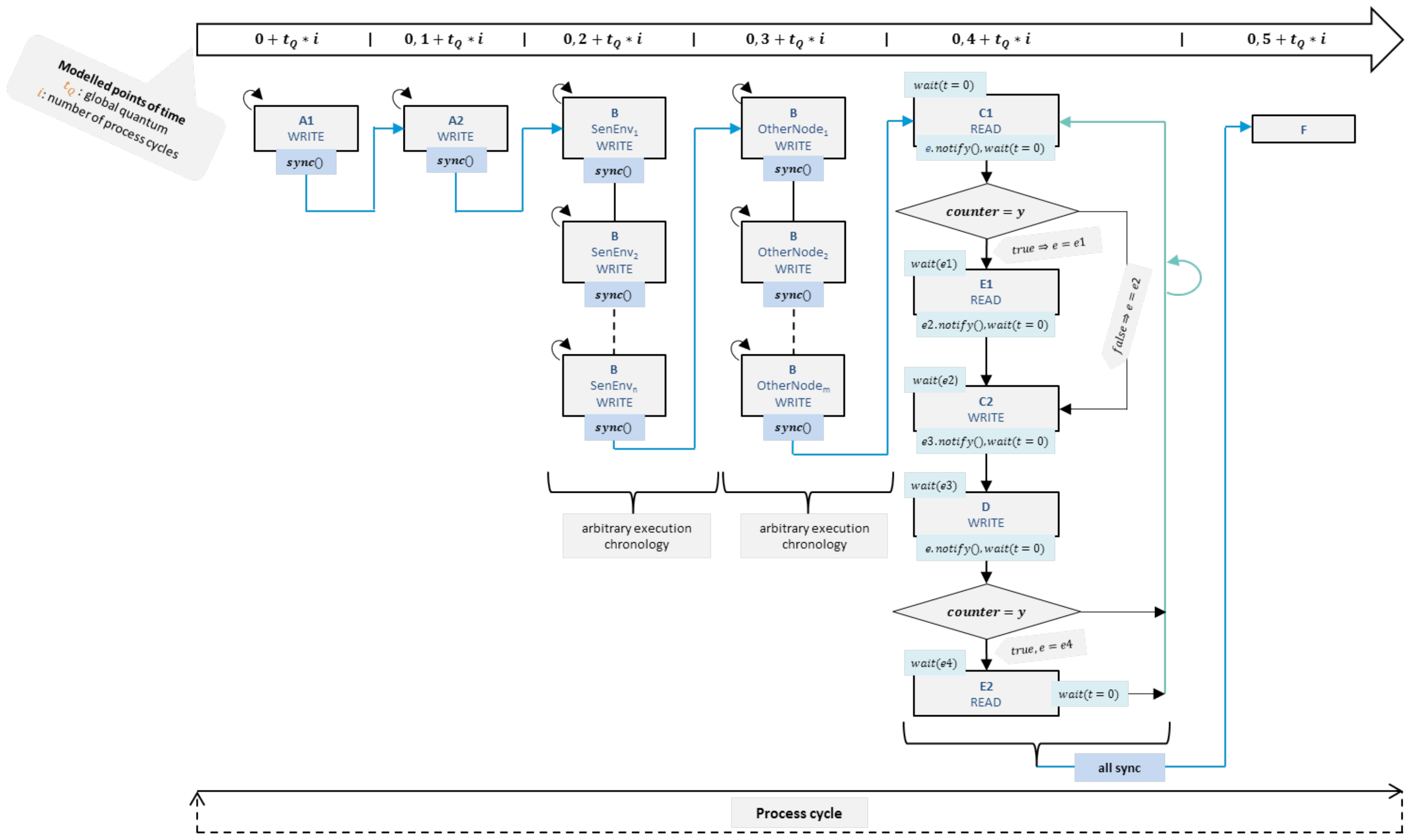


Figure 2.12: *Implemented chronology of processes.*

when the scheduler has advanced the simulation time of this same local time offset t_{off}. From that statement, 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. So: $t_{sim,next} = t_{sim,actual} + t_{off}$.

The local time offset, in turn, results from the sum of latency times of all the transactions generated by a process between two synchronization points. In this model, the latency times and the number of transactions between the synchronization points of a process are automatically calculated during elaboration such that the local time offset between two synchronization points always equals to the global quantum. Therefore, the next execution time of a process after his last synchronization always results from the sum of the actual simulation time and the specified global quantum $t_{glob\_quantum}$. So,

$$t_{sim,next} = t_{sim,actual} + t_{glob\_quantum}.$$

From the known simulation time of the first execution of each process (t_{beg}) and the above formula, we have the following start time for the execution of a process
in the 1st process cycle:

$$t_0 = t_{beg}$$

in the 2nd process cycle:

$$\begin{aligned} t_1 &= t_0 + t_{glob\_quantum} \\ &= t_{beg} + t_{glob\_quantum} \end{aligned}$$

in the 3rd process cycle:

$$\begin{aligned} t_2 &= t_1 + t_{glob\_quantum} \\ &= (t_{beg} + t_{glob\_quantum}) + t_{glob\_quantum} \\ &= t_{beg} + 2 * t_{glob\_quantum} \end{aligned}$$

in the $4th$ process cycle:

$$\begin{aligned} t_3 &= t_2 + t_{glob\_quantum} \\ &= (t_{beg} + 2 * t_{glob\_quantum}) + t_{glob\_quantum} \\ &= t_{beg} + 3 * t_{glob\_quantum} \end{aligned}$$

So we have the following start time in the nth process cycle

$$t_{n-1} = t_{beg} + (n-1) * t_{glob\_quantum}, \text{ mit } n \in N^+ - \{0\}$$

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:

$$\boldsymbol{t_i = t_{beg} + i * t_{glob\_quantum}}, \text{ mit } \boldsymbol{i = n - 1} \text{ und } \boldsymbol{n \in N^+ - \{0\}}$$

At this point, it is also important to mention that the predetermined start times t_{beg} for the first execution of the respective processes weren't defined arbitrary. They have been kept as *small* as possible in order to avoid them having a decreasing effect on the number of transactions, which should be executed in each process cycle and which is calculated during elaboration. This in turn would lead to data inconsistencies.
For example, if process C1 generates fewer read transactions in a process cycle than calculated, then some sensor data stored in the component SAE would neither be readout nor processed at all during the simulation. For processes A1 and A2, fewer write transactions than expected would mean that some of the g.v.o.c. data won't be transferred to the monitor and the self-expressive engine. This would prejudice the node functionality.

2.4.4.2 Process chain C1-[E1]-C2-D-[E2]

As already described in the section 2.4.3, each process of the process chain is suspended after each completed transaction by calling the function wait() with SC_ZERO_TIME as time argument. Immediately prior to this function call, en event belonging to the dynamic sensitivity of the next process in the chain, is notified (immediate notification).

Process C1 is the first one in the chain. Thus, it doesn't have any event in his dynamic sensitivity. The execution of all other processes in the process chain depends on its immediate notification of the event belonging to the dynamic sensitivity of the next process (either E1 or C2). Every other process in the chain is then run after the previous one has notified his event and has suspended.
Because the execution order of processes within a delta cycle can not be controlled by the implementation, we have to make sure that process E1 or C2 never miss the immediate notification of their respective events ($e1$ or $e2$), which is performed by process C1. As a matter of fact, if process C1 is run before C2 in a delta cycle, it will notify $e2$. But C2 will miss this immediate notification, because it won't be executing the following statement $wait(e2)$. The same applies for process E1 and its event $e2$.
Therefore, the statement *wait(sc_zero_time)* has been added before all operations of process C1 already cited. So, within each quantum and before each transaction, process C1 first executes a delta notification and other processes run until their respective $wait(e)$ statement, at which point they are also suspended. In the next delta cycle of the same quantum, process C1 notifies either $e1$ or $e2$ and suspend. The corresponding and already awaiting process is therefore executed and, in turn, triggers the execution of it follower by also notifying the corresponding event.

This is how the execution chronology of process in the chain is ensured in each process cycle. This chronological order was defined in accordance with the given definition of the proprioceptive node.

2.4.5 Key parameters

As key parameters in the model, we describe every parameter that contributes to ensure the implemented time decoupling of processes and to the correct addressing of the memory areas during transactions, and thereby play an important role for the deterministic behaviour of the model during simulation as well as for the correct functionality of each node.

There are two types of key parameters to be distinguished: the user parameters which are given by the user before simulation and the derived parameters which are determined during elaboration using the user parameters. Both parameters are presented in the following sections. For the user parameters, the limits are given and for the derived ones, the used function is derived.

2.4.5.1 Number of components

As already mentioned, a proprioceptive node can have many sensors as well as many actuators. The number of components is used here to designate the number of sensors and actuators, i.e. the number of TLM components SenEnv, OtherNode, Actuator and ExtAction) in a node, whose count depend on the types of information to be processed in a node, the complexity as well as the topology of the proprioceptive system to be simulated. These values must be specified by the user for each node and before simulation begin as positive integers so that the required number of TLM components are generated or rather instantiated with respect to the made specifications.

2.4.5.2 Number of dataset per process-cycle and length of a dataset

The term *dataset* is used here to describe a record of data that belongs together or that need to be processed together.
In order to ensure the correct processing of the sensor and g.v.o.c. data within each node, the user must specify the maximal length of a dataset (DL_{max} in $Bytes$) and the number of datasets (as a positive integer) that should be process within each process cycle.

2.4.5.3 Memory areas or data storage objects

There are multiples storages objects in the node model. But particularly important here is the storage object of the TLM component SAE, because it can be simultaneously accessed (write access) by many sensors within a process cycle and each with a different type of information. So, the implementation must prevent that different sensors overwrite one another's data within a process cycle. A further argument is that each TLM component SenEnv or OtherNode can initiate many transactions within a process cycle, in other words, it can write more than one dataset on the storage object of the TLM component sae within a process cycle. Here again the implementation must prevent a sensor from overwriting his own datasets within a process cycle. Finally, there must be enough data storage space on the TLM component SAE for all the sensor data to be stored within each process cycle.

This is why the required size G_{SAE} of the data storage object owned by the TLM component SAE is determined during elaboration. For this purpose, we use the following formula:

$$\boldsymbol{G_{SAE} = (n_{SE} + n_{ON}) * n_D * DL_{max}}$$

where

G_{SAE}	denotes the size of the SAE-memory in Bytes,
n_{SE}	denotes the number of the SenEnv components,
n_{ON}	denotes the number of the OtherNode components,
n_D	denotes the number of datasets of each sensor pro process-cycle and
DL_{max}	denotes the maximal length of a dataset in Bytes

For the correct addressing of this memory area and in order to prevent the overwriting scenarios described above, the transaction address, which is given by every sensor and which indicates the start address of the write operation, is mapped to the address range of this storage object by the interconnect component for each method call and indeed before the call is forwarded. For this purpose, the following linear mapping function is used:

$$\boldsymbol{f(adr) = (n_D * DL_{max}) * id_{ts} + adr}$$

where

adr	denotes the given start address for the transaction,
id_{ts}	denotes the identification number of the tagged method call,
n_D	denotes the number of datasets of each sensor pro process-cycle and
DL_{max}	denotes the maximal length of a dataset in Bytes

In addition to the storage object of the TLM component SAE, it exists two further data storage objects each owned by the TLM components MONITOR and SEE and accessed respectively by the processes A1 and A2 of the TLM component GVOC. As already seen, these processes can run many times within a process cycle and each of them generates a write transaction by every execution. Here also, the implementation must prevent the TLM component GVOC for overwriting his data on this stated storage objects and must also provides enough memory space for all the data to be stored within a process cycle. This leads us to the following formula for the size of both storage spaces:

$$G_{\{MONITOR,SEE\}_{GVOC}} = n_{D_{GVOC}} * DL_{max_{GVOC}}$$

where

$G_{\{MONITOR,SEE\}_{GVOC}}$	denotes the size of the data storage space for GVOC data,
$n_{D_{GVOC}}$	denotes the number of g.v.o.c. dataset to be transferred within a process cycle and
$DL_{max_{GVOC}}$	denotes the maximal length of a g.v.o.c. dataset.

For all other storage objects, i.e. the report storage objects of the LModel and the SEE components, the storage object of the SEE for the decision data of the self-aware component and lastly the storage objects of the internal and external actuators, the required size (G_{OTHERS}) is always equals to the maximal length of a dataset (DL_{max}) to be stored, because each dataset stored must only be available for the current delta cycle. This length must be specified by the user (in Bytes) before the simulation start. So:

$$\boldsymbol{G_{OTHERS} = DL_{max}}$$

2.4.5.4 Global quantum

When using temporal decoupling, it is recommended that each process use the same global time quantum. Furthermore, it is proven in [**?**, p. 269 and ff.] that the simulation performance depends on the value of the global quantum. It should be determined, in accordance with the whole simulation time period, so that the number of resulting synchronizations or delta cycles doesn't exceed a few hundred thousands (see [**?**, p. 279]).

Let's assume that t_{sim} denotes the whole simulation time period, n_{delta_cycles} the number of delta cycles and, lastly, n_{sync} the number of synchronizations. The number of synchronizations in the model can be calculated with the following formula:

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

From the execution chronology of processes previously described, it arises that each of the processes A1, A2, B's executes all its transactions within a delta cycle in a quantum. As for the processes of the processes of the process chain, each of them needs two delta cycles in order to perform a transaction within each quantum. Finally process F always needs a delta cycle within each quantum. The following figure illustrates the delta cycles between every two successive synchronization points or rather within each quantum in a node during the simulation.

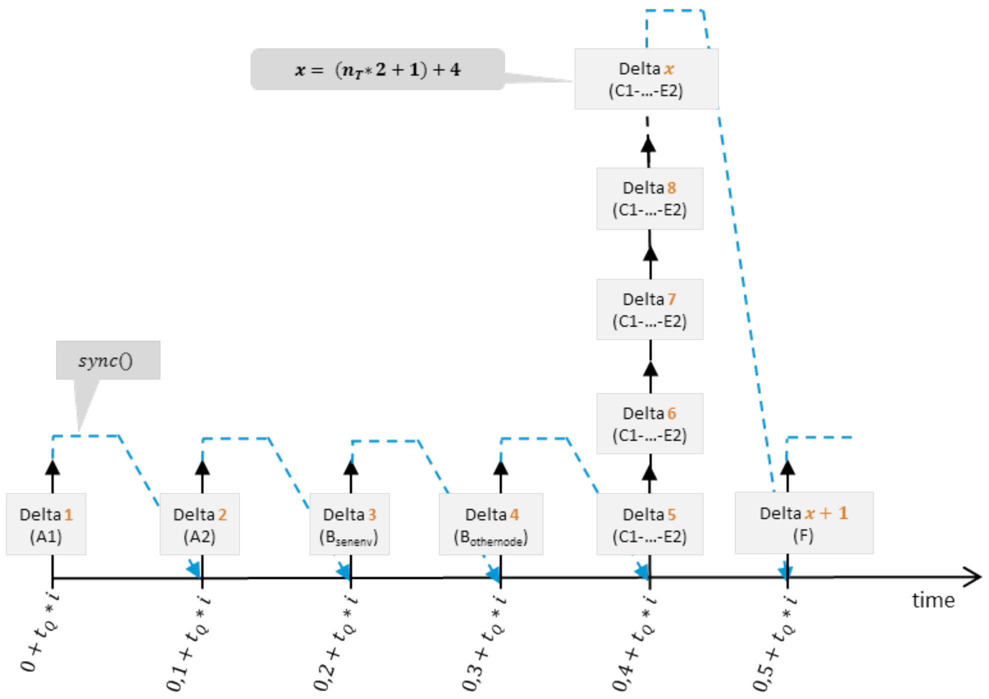


Figure 2.13: *Delta cycles between the synchronization points in a node.*

Using both, the illustration above and the previous formula for the number of synchronizations, we can derive the formula for the number of delta cycles in this 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_T denotes the number of generated transactions by every process in the chain and $n_{nodes\_nr} \geq 1$ denotes the number of nodes in the simulated system.

Thus, the following inequations should hold:

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

or

$$\begin{aligned} n_{delta\_cycles} &\leq 100000 && \Leftrightarrow \\ \left\lfloor \frac{t_{sim}}{t_{glob\_quantum}} \right\rfloor * (6 + n_T * 2) * n_{nodes\_nr} &\leq 100000 && \Leftrightarrow \\ \left\lfloor \frac{t_{sim}}{t_{glob\_quantum}} \right\rfloor &\leq \frac{100000}{(6+n_T*2)*n_{nodes\_nr}} \end{aligned}$$

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

$$\left\lfloor \frac{\boldsymbol{t_{sim}}}{\boldsymbol{t_{glob\_quantum}}} \right\rfloor \leq \frac{\mathbf{100000}}{\mathbf{(6 + \boldsymbol{n_T} * 2) * \boldsymbol{n_{nodes\_nr}}}}$$

2.4.5.5 Latency times of the transactions

From the synchronization condition for temporally decoupled processes given in section 2.4.2, 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:

$$t_{glob\_quantum} \leq \sum_{i=1}^{n_T} t_{trans\_delay,i} = t_{off}$$

where

$t_{glob\_quantum}$	denotes the given global quantum,
n_T	denotes the number of transactions,
$t_{trans\_delay,i}$	denotes the latency of the *ith* transaction and
t_{off}	denotes the local time offset

To determine the execution start times of the processes in the process cycles, we assume that the value of time local offset is equal to the global quantum. So $t_{glob\_quantum} = t_{off}$.

If every transaction has the same latency, then the sum of the latency times in the inequation above can be substituted by the product obtained when multiplying the latency of one transaction by the number of executed transactions between two synchronization points. So:

$$t_{glob\_quantum} \leq n_T * t_{trans\_delay}$$

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. It is:

$$t_{trans\_delay} = t_{glob\_quantum}/n_T$$

where

n_T	denotes the number of transaction per process-cycle

Given the fact that all sensor data stored on the TLM component SAE within a process cycle must also be readout and evaluated by the TLM component LMODEL within the same process cycle, the number of transactions generated by each of both processes C1 and C2 can be calculated as follows:

$$n_{T_{C1,C2}} = n_{T_B} * n_S$$

where

$n_{T_{C1,C2}}$	denotes the number of transactions generated in each process cycle by C1 and C2,
n_{T_B}	denotes the 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 must run in each process cycle as many times as the processes C1 and C2.

By substituting the number of transactions determined above 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 of 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 ($BL > 1$). 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:

$$t_{trans\_delay} = t_{single\_delay} * BL \text{ mit } BL = \left\lceil \frac{DL_{max}}{(BUSWIDTH/8)} \right\rceil$$

This leads to the following formula for the latency of the single transactions in the node:

- for the processes A1, A2 and B's:

$$\boldsymbol{t_{single\_delay} = t_{glob\_quantum}/(n_T * BL)}$$

- for the processes C1, C2, E1, E2 and D of the process chain:

$$\boldsymbol{t_{single\_delay} = t_{glob\_quantum}/(n_{T_B} * n_S * BL)}$$

These are the formulas implemented in the model to automatically calculate the latency times of the single transactions in each node of a simulated proprioceptive system, as mentioned before. This calculation happens during elaboration with the following user specifications: the global time quantum, the number of sensor datasets to be processed within a process cycle, the number of sensors and lastly the maximal length of the respective datasets to be processed. This ensures the preconditioned execution chronology of the processes and transactions in each process cycle as well as the respective execution start times of the processes that we defined.

2.4.6 Transaction flows

By the non-negligible number of processes in the model of a proprioceptive node and given the fact that this number increases proportionately to the number of nodes in the simulated system, it can be difficult to track the consequently increasing number of generated transactions in the model. The sequence diagrams below which illustrate the transaction flows of the processes should help in this issue.

The following figure illustrates the transaction flows of the processes A1, A2, and B's in each process cycle:

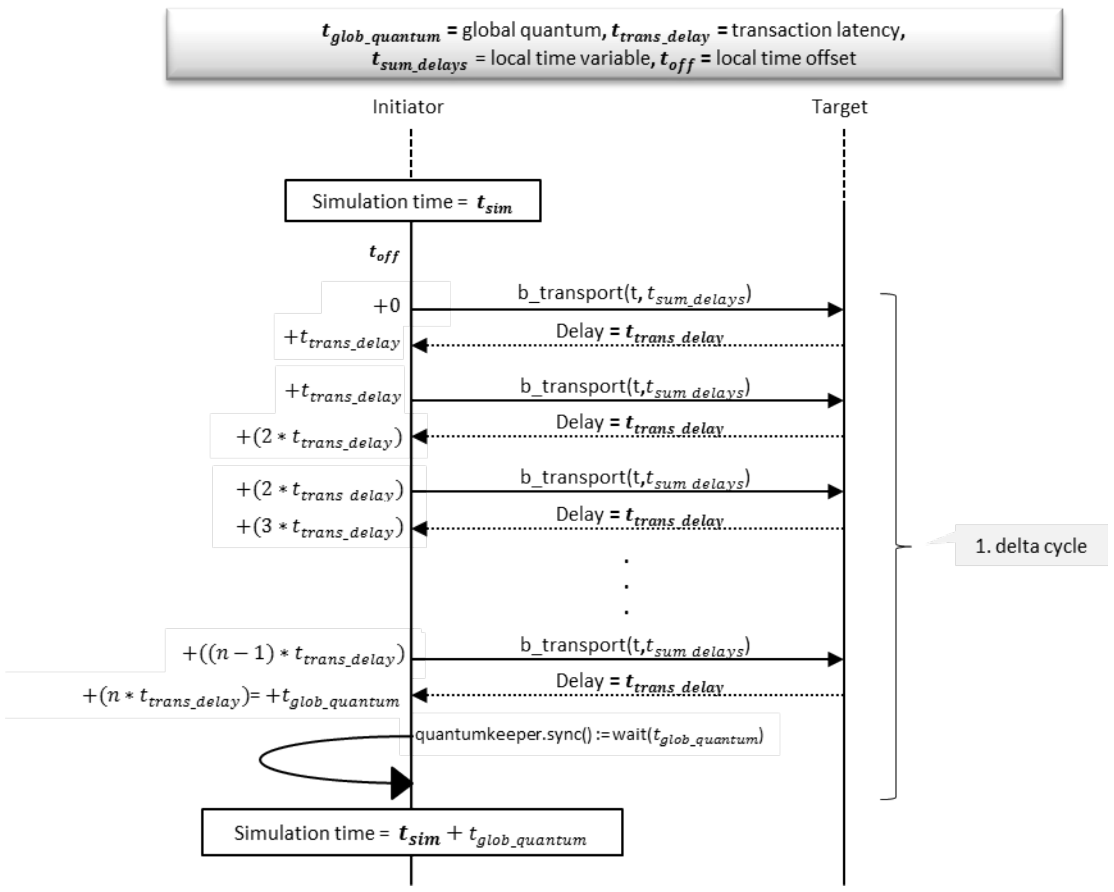
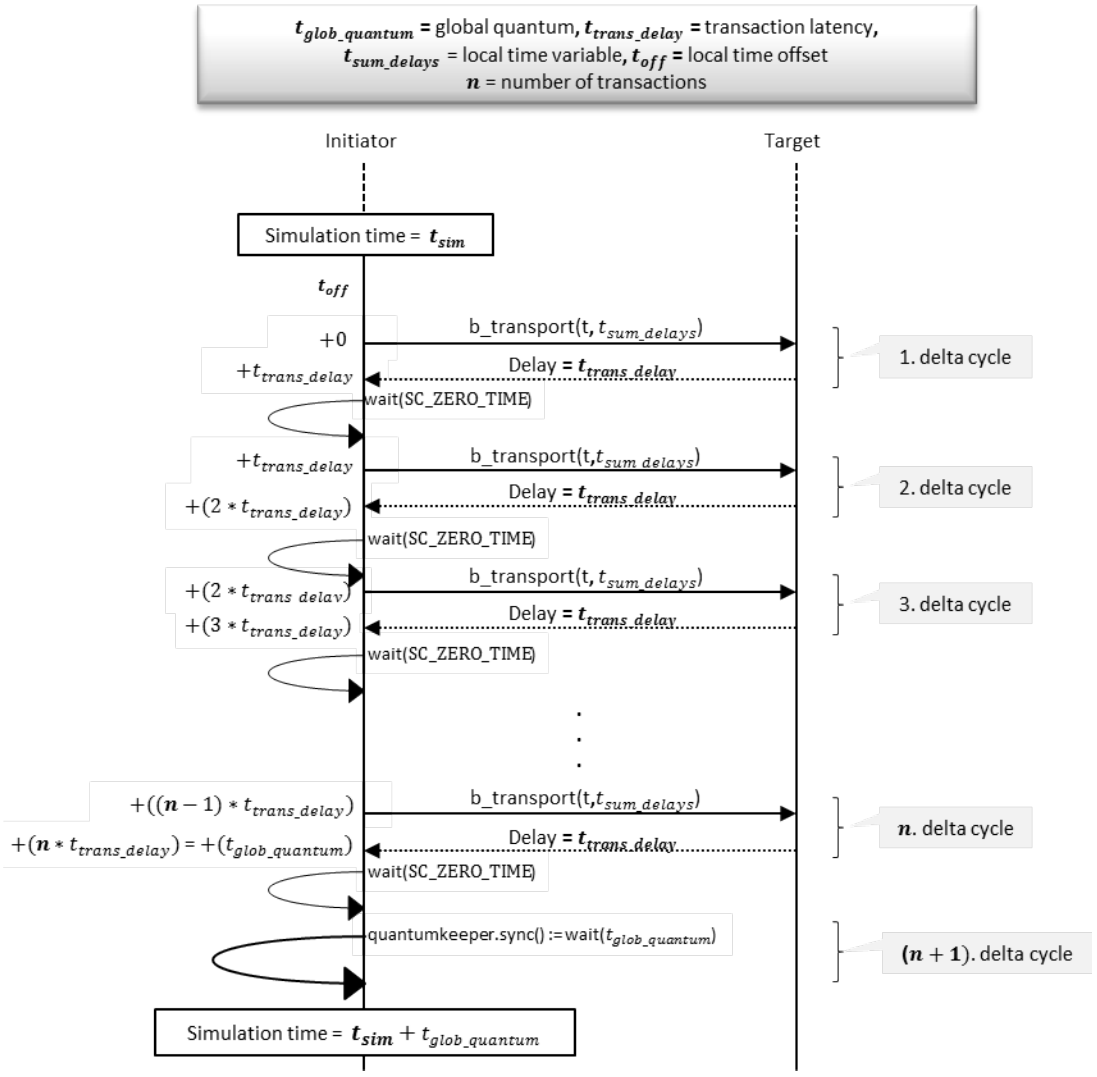

$t_{glob\_quantum}$ = global quantum, t_{trans_delay} = transaction latency,
t_{sum_delays} = local time variable, t_{off} = local time offset
Initiator
Target
Simulation time = t_{sim}
t_{off}
$+0$
b_transport(t, t_{sum_delays})
$+t_{trans\_delay}$
Delay = t_{trans_delay}
$+t_{trans\_delay}$
b_transport(t,t_{sum_delays})
$+(2 * t_{trans\_delay})$
Delay = t_{trans_delay}
$+(2 * t_{trans\_delay})$
b_transport(t,t_{sum_delays})
$+(3 * t_{trans\_delay})$
Delay = t_{trans_delay}
1. delta cycle
$+((n-1) * t_{trans\_delay})$
b_transport(t,t_{sum_delays})
$+(n * t_{trans\_delay}) = +t_{glob\_quantum}$
Delay = t_{trans_delay}
quantumkeeper.sync() :=wait($t_{glob\_quantum}$)
Simulation time = $t_{sim} + t_{glob\_quantum}$

Figure 2.14: *Transactions flows for the processes A1, A2, B in each process cycle.*

The transaction flows for each process (C1, C2, E1, E2 and D) of the process chain are illustrated in the following figure.



$t_{glob\_quantum}$ = global quantum, t_{trans_delay} = transaction latency,
t_{sum_delays} = local time variable, t_{off} = local time offset
n = number of transactions
Initiator
Target
Simulation time = t_{sim}
t_{off}
$+0$
b_transport(t, t_{sum_delays})
$+t_{trans\_delay}$
Delay = t_{trans_delay}
1. delta cycle
wait(SC_ZERO_TIME)
$+t_{trans\_delay}$
b_transport(t,t_{sum_delays})
$+(2 * t_{trans\_delay})$
Delay = t_{trans_delay}
2. delta cycle
wait(SC_ZERO_TIME)
$+(2 * t_{trans\_delay})$
b_transport(t, t_{sum_delays})
$+(3 * t_{trans\_delay})$
Delay = t_{trans_delay}
3. delta cycle
wait(SC_ZERO_TIME)
$+((n-1) * t_{trans\_delay})$
b_transport(t,t_{sum_delays})
$+(n * t_{trans\_delay}) = +(t_{glob\_quantum})$
Delay = t_{trans_delay}
n. delta cycle
wait(SC_ZERO_TIME)
quantumkeeper.sync() :=wait($t_{glob\_quantum}$)
$(n+1)$. delta cycle
Simulation time = $t_{sim} + t_{glob\_quantum}$

Figure 2.15: *Transaction flows for the processes C1, C2, E1, E2 and D per process cycle.*

As for the synchronization process F, which is executed in each node and in each process cycle only once and doesn't generate any transactions, it not possible to draw its sequence diagram. There is indeed no transaction to represent.