From Model-Based to Real-Time Execution of Safety-Critical Applications: Coupling SCADE with OASIS*

Simon Bliudze†
EPFL
RiSD, Station 14
CH-1015 Lausanne, Switzerland
Simon.Bliudze@epfl.ch

Xavier Fornari
Esterel Technologies
Parc Euclide, 8 rue Blaise Pascal
Elancourt, 78996 France
Xavier.Fornari@esterel-technologies.com

Mathieu Jan
CEA, LIST
Point Courrier 94
Gif-sur-Yvette, F91191 France
Mathieu.Jan@cea.fr

Abstract

Developing embedded safety critical real-time systems and ensuring properties such as deterministic behaviour in a simple way for the application designers is a challenging task. A large number of commercial and academic real-time operating systems (RTOS) as well as model-based development environments based on synchronous languages are available. Automatic transformations from synchronous modelling languages to RTOS are important for streamlining development of real-time applications without compromising the guarantees of their safety. In this paper, we present an automatic transformation from the SCADE synchronous language into applications for the OASIS safety-oriented real-time execution platform, a multi-scale time-triggered approach. This transformation has been partially implemented and we illustrate it with an industrial case-study from the domain of medium voltage protection relays.

1 Introduction

Various domains, as for instance the automotive or the avionic industries, develop increasingly complex real-time systems. Model-based approaches have been proposed to specify requirements of such systems and design them. Such design environments allow modelling and simulation of applications as well as generation of qualifiable/certified code. Generated code can then be compiled and run on various Real-Time Operating Systems (RTOS), such as Integrity, VxWorks or PikeOS. Such RTOS are based on an event-triggered approach for the execution of applications and do not provide sufficient predictability and analyzability: the deterministic behaviour of an application cannot be ensured prior to its execution.

The goal of the OASIS [4] approach is to build safety-critical real-time systems where the system behaviour is independent from the asynchrony that is allowed during the execution of an application. The system behaviour is therefore unvarying, unique, and independent from its realization on a target computer. Therefore and by construction, OASIS is a complete answer to demonstrate the system timeliness: all timing constraints of all activities are clearly expressed in the design phase and can be formally proven to satisfy (or not!) the capacities of the hardware support. OASIS is available on various architectures and is currently in use in industrial products [3].

SCADE is a synchronous language derived from Lustre [7] and implemented in the Esterel Technologies SCADE Suite® model-based development environment dedicated to critical embedded software.

To the best of our knowledge, few connections exist between synchronous modelling languages, which allow one to develop applications with certified functional behaviour, and time-triggered execution platforms, which guarantee temporal determinism of the systems. Our contribution, in this paper, is 1) the description of an automatic transformation from SCADE models to OASIS applications and 2) an illustration of this coupling on an industrial case study.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 discusses

*This work was partially supported by the FP7 European project INTERESTED (grant agreement 214889).

†Work realised while this author was with CEA LIST.

the notions of real and logical time and provides OASIS background. Section 4 presents our SCADE to OASIS transformation, in particular the computation of OASIS clocks from the SCADE model of an application. Section 5 presents the industrial case study. Finally, Section 6 concludes and suggests directions for further research.

2 Related work

The context of our work is closest to that of a code generation tool-chain [1] from SCADE/Lustre models for the Time-Triggered Architecture (TTA) [12]. TTA is a distributed, synchronous, fault-tolerant architecture for a set of nodes connected through a bus and exchanging data using a time triggered protocol. In [1], Lustre has been extended with additional primitives to specify code distribution, timing requirements and deadlines. Note that our work could leverage extension of OASIS to a distributed bus-based architecture [2], where scheduling is computed transparently for the application developer [9].

An extension of Simulink to express designs of the time-triggered Giotto language is presented in [11], where I/O timing is generated automatically. Then, a schedulability analysis is performed by the Giotto compiler and code is generated for the HelyOS or OSEKWorks RTOS. This tool-chain has been demonstrated on a helicopter autopilot system.

A mapping of synchronous models onto the Loosely Time-Triggered architecture (LTTA) maintaining the semantic equivalence between the synchronous model and its implementation over LTTA is presented in [14]. No case study is considered. The LTTA architecture assumes that the Communication by Sampling (CbS) paradigm is implemented over a bus and does not require a clock synchronization mechanism. Note that OASIS temporal variables communication mechanism is more powerful than CbS and provides guarantees on the freshness of the data being used. Additionally, LTTA assumes that each node of the architecture runs a single process in a periodic manner.

A compilation chain for generating from a multi-periodic synchronous language tasks that can be executed on a real-time platform with a dynamic-priority scheduler (EDF) is presented in [6]. The dependencies between tasks are reduced to independant tasks by adjusting task release dates and deadlines.

3 Background

3.1 Real and Logical Time

When developing real-time systems, it is important to make a clear distinction between real and logical time. The notion of time serves two purposes. Firstly, it is used to specify the order of execution of individual actions. Secondly, it can be used to specify the durations. Logical time can be used in both cases, whereas specifying the order of execution based on the real time leads to the non-determinism of execution, as real time is not known at the design stage.

One of the fundamental characteristics of synchronous languages (e.g. Lustre, Scade, Signal) resides in the use of clocks for the specification of the synchronization points between the components. A clock is an infinite subset of natural numbers. The simplest example of a synchronous system consists of a number of components all referring to the same periodic clock, defined by its initial date and its period. The operational semantics of such a system is defined as a sequence of cycles executed at each tick of the clock. A cycle consists of three phases: acquisition of inputs, computation and publication of outputs. The actual (real time) duration of computations is then irrelevant. This is materialised by the so-called synchronous hypothesis: computations are assumed to have no duration or, in other words, the computation duration is negligible compared to that of communication among the components. This leads to computation being divisible in steps and execution being well-behaved. Real-time performance is then evaluated by first computing the bounds or estimations of worst-case execution time (WCET) of individual computations, then performing an end-to-end delay analysis of the entire system [5].

Although recent advances in WCET analysis [15] allow system designers to obtain more precise results for the individual computations, the end-to-end delay analysis of concurrent systems remains complex. Furthermore, when end-to-end delays do not satisfy the real-time constraints of the system, the key factors have to be identified, re-designed and the whole analysis process restarted from the beginning. The opposite, declarative approach adopted, among others, in OASIS is the so-called *time-triggered approach.* This consists in assigning to each action its *desired* execution time guaranteeing by construction the end-to-end constraints. The system is then implemented over an execution kernel running on a bare-bone platform and ensuring that 1) the action currently being executed does not exceed the duration it is allocated and 2) the execution of the following action is triggered at the appropriate real time. Various approaches can be taken in the case when an action violates a deadline: execution can aborted, a recovery mechanism or a degraded mode can be initiated. Alternatively, WCET analysis can be performed in order to verify whether execution time requirements are respected and select an appropriate target architecture. In both cases, this avoids the complex repeated analysis described above for the traditional approach.

Finally, with the time-triggered approach, exe-

cution of an action must not start before the real time date of the corresponding logical instant, as this could compromise data coherence among components guided by the logical time. Thus time-triggered approach reconciles the real and logical times.

3.2 OASIS: A Framework for Safety-Critical Real-Time Systems

The main objective of OASIS is a framework encompassing models, methodologies, and tools for the development of embedded critical software exhibiting completely deterministic temporal behaviour. This objective is achieved by adhering to the following two fundamental principles:

1. OASIS systems are based on a time-triggered architecture,
2. communication between tasks is non-blocking and time-stamped (including the visibility and expiry dates of the communicated elements).

The second principle ensures that all the synchronization is limited to that between individual tasks and the kernel, thus guaranteeing that each elementary action executes within the temporal window it is assigned at the design time. Hence, the actual implementation of a system respects its formal semantics defined by the OASIS computation model. Consequently, the fact that visibility and expiry dates are explicitly specified for all communicated elements (messages and values of shared variables) results in complete temporal determinism of the system behaviour. The sequencing of activities is independent from the its realization on a target computer.

OASIS implementation comprises a programming language PsyC (Parallel synchronous C), which is an extension of C. The extension allows one to specify tasks and their temporal constraints as well as their interfaces. The computation model of OASIS relies on a collection of communicating tasks (a specific flavour of timed automata) with the operational semantics defined using a collection of logical clocks. OASIS implements a multi-scale time-triggered execution model, in which a collection of execution graphs represent the states of the tasks and the deadlines for the corresponding set of statements of the tasks. An OASIS task is called an *agent*. An OASIS system can be executed on a bare-bone target machine (mono-processor, multi-core or distributed) or on a POSIX platform (with real or simulated time). All types of execution, except of the execution in simulated time, are behaviourally and temporally deterministic and equivalent.

3.3 OASIS: Data-flow Communication

Two communication mechanisms are available in OASIS: 1) a 1-to-n regular flow of values called *temporal variables* and 2) an n-to-1 irregular exchanges through messages. The new values of a temporal variable are made visible at every synchronisation point of the producer task, while messages require explicit definition of visibility dates. SCADE signals correspond to temporal variables in OASIS. Hence OASIS messages are out of the scope of this paper.

Each temporal variable is declared in the interface of a single producer agent, but can be consulted by as many agents as necessary. Below is a PsyC example with one producing and one consulting agent.

```
agent AG_A (... /* start time */) {
 temporal {double 0$x = 0.0;}
 display {x : AG_B;}
  ...
}
agent AG_B (... /* start time */) {
 consult {AG_A : 2$x;}
 ...
}
```

The clause `temporal` in `AG_A` declares a temporal variable `x` of type `double`. The clauses `display` in `AG_A` and `consult` in `AG_B` specify that the latter is authorised and will consult the values of `x`. The expressions `0$` and `2$` preceding the name variable in clauses `temporal` and `consult` declare the number of past values of `x` that the corresponding agents require. The past values of temporal variables are stored in the buffers maintained by the OASIS system layer and generated automatically. The size of each buffer is statically determined by the PsyC compiler.

4 From SCADE to OASIS

OASIS guarantees the temporal and behavioural determinism of the implementation of a system. In other words, it guarantees that the operational semantics of the computation model is respected. When an OASIS system is obtained by a translation from another formalism, such as SCADE, one has to ensure that the OASIS semantics of the generated model is equivalent to that of the initial model in the source formalism. This is achieved by defining the clocks and the temporal synchronization points determining the execution windows for all elementary actions of the agents in the system. The order of execution of elementary actions must respect the causality relation defined by the logical clocks in the initial system described in the source formalism.

The SCADE/OASIS transformation is implemented as a SCADE Suite adaptor. An adaptor is a software component that can be added to the SCADE Suite environment and that is called when generating code or building an application from a SCADE model. In addition to the classical C code generated for all SCADE operators, the adaptor provides the appropriate encapsulation in PsyC of the C functions

generated from the SCADE model. C code is generated for all SCADE operators as usual.

The SCADE/OASIS transformation adaptor analyses the hierarchical model to extract a representation of the network of operators. PsyC code is generated only for those operators that have an actual behaviour and are not intermediate connection levels. Dedicated agents are also generated for the **fby** (*followed-by*) delay operators found in the network. The interconnection of the PsyC agents is equivalent to the initial network. PsyC agent `AG_A` generated for a SCADE node A is schematically shown below.

```
agent AG_A (starttime = 1 with C_A) {
  ... /* agent connection interface */
  body start {
    A_reset (&outC);
    next main;
  }
  body main {
    A (..., &outC);
    ... /* update of the produced outputs
           based on the values in outC */
    /* Advance 1 step on the C_A clock */
    advance (1) with C_A;
  }
}
```

In this scheme, `A_reset()` and `A()` are C functions generated by the SCADE Suite code generator for the initialisation of the node and for the execution of one computation round respectively. The arguments passed to these functions are the input values of the agent and the structure `outC` containing the context of the node (local variables and output values). We omit the agent elements defining the interface for interconnection with other agents.

OASIS operational semantics of this agent is the following: 1) at the first instant of the logical clock `C_A`, the agent computation is initialised, in the body `start`, by a call to the function `A_reset()`; 2) it then immediately moves to the body `main`; 3) the agent calls the function `A()`, updates the output and advances one step on the clock `C_A`. The last step is repeated indefinitely.

The clock `C_A`, in combination with clocks of other agents, defines the synchronisation points where data is exchanged. Hence, it is the key element defining the semantics of the application in this transformation scheme. In the following sections we explain how these clocks are computed and discuss the semantic equivalence of a SCADE model with the corresponding generated OASIS application.

4.1 Computation of the Agent Clocks

The main difficulty in the SCADE/OASIS transformation is computing the logical clocks that define the temporal behaviour of the agents in such a way

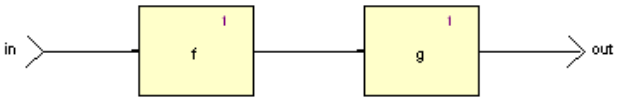


Figure 1. Simple composition of two operators in SCADE.

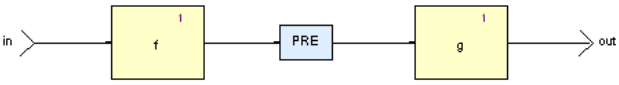


Figure 2. Composition with a pre operator.

as to preserve the functional semantics of the model and optimize the usage of computing resources.

One of the main differences between SCADE and OASIS computation models concerns the propagation of data during one computation cycle. In SCADE, data is propagated from inputs to outputs at each computation cycle. Consider the SCADE operator in Fig. 1. At a given cycle n, the node f reads the input in_n and computes an output x_n, the node g then immediately reads this value and produces an output out_n. Supposing that neither f nor g contain delays, we have $out_n = g(x_n) = g(f(in_n))$.

In OASIS, all agents read their inputs simultaneously and then simultaneously publish the results of their computation. Taking on the above example, at the cycle n, both f and g would read their respective inputs in_n and x_{n-1} (the output produced by f at the end of the previous cycle) and produce the corresponding outputs $x_n = f(in_n)$ and $out_n = g(x_{n-1}) = g(f(in_{n-1}))$. This can be translated by the following intuition: a simple OASIS system corresponds to a network of SCADE nodes with a **pre** operator on each communication link (Fig. 2).

In order to preserve the computation semantics throughout the SCADE/OASIS transformation, each agent is assigned a specific slot of the computation cycle. Thus, the intermediate values are computed and published by the corresponding agents before the agents that require these values for their computations start their respective cycles. This is illustrated in Fig. 3 with the green lines identifying the original cycles of the SCADE model and the dashed arrows—the data flow.

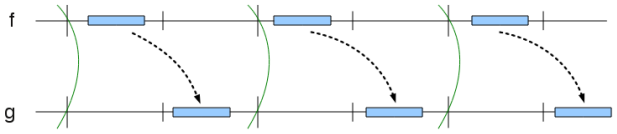


Figure 3. Execution window configuration for agents in basic transformation.

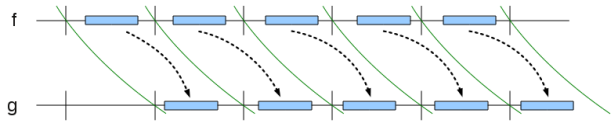


Figure 4. Optimised execution window configuration.

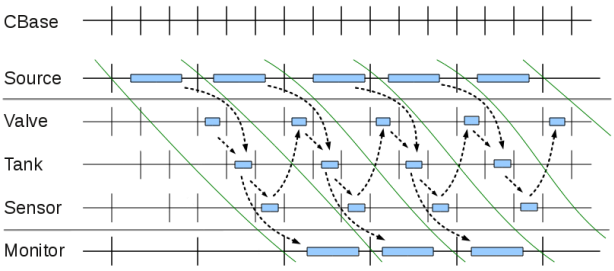


Figure 6. Optimised OASIS coordination for the SCADE model in Fig. 5.

It is clear that this implementation is not optimal in terms of parallelism. An alternative semantically equivalent implementation is shown in Fig. 4.

Consider now a slightly more complex example of level control in water tank (Fig. 5). (Monitor is an imported node used to observe all the variables in the system during a simulation.)

The circular dependency between Tank, Sensor and Valve nodes does not violate SCADE semantic constraints due to the presence of the delay (**fby**) operator, but it prevents us from using the same optimisation as in the previous example. Indeed, in SCADE semantics, the values produced by all components must be computed before the next cycle can begin. These tree nodes form a *strongly connected component* in the data dependency graph of the application. The other two strongly connected components are singletons, i.e. consist of one node each: Source and Monitor respectively. On the other hand, the above optimisation can be applied *among* the strongly connected components (Figure 14). In addition to the elements shown in previous figures, here we add a base clock $CBase$ used to derive the logical clocks defining the execution windows in the OASIS model. In order not to overcharge the figure, we do not show all data dependencies in this application.

A general method to compute an optimised semantics preserving execution windows arrangement for the SCADE to OASIS transformation consists of the following steps:

1. Compute the strongly connected components of the application data dependency graph.

2. The factorisation of the application data dependency graph, that is the data dependency graph among the strongly connected components, is a *directed acyclic graph* (DAG). This allows one to compute the depth of each strongly connected component.

3. All strongly connected components run in parallel with the same temporal window corresponding to one cycle of execution of the SCADE model, but with the start delayed by their depths as defined in step 2.

4. For each component, we break the cycles on the delay operators (pre, fby), thus also obtaining a DAG. To each node of the component, we assign its depth in this DAG.

5. For each component, we break its associated temporal window into n slots, where n is the depth of the entire DAG associated to this component (maximal depth of the nodes plus one), and associate to each node the execution window corresponding to the slot numbered by its depth in the DAG.

An implementation of this construction with OASIS clocks relies on a base clock refining all involved slots. Such a base clock can be straightforwardly computed as follows. Let P be the real-time duration associated to one cycle of synchronous execution in SCADE, and let N be the least common multiplier of the depths of all strongly connected components above. We fix the base clock $CBase$ with the start time 0 and period P/N. To each strongly connected component Cmp, with the associated DAG of depth n, we associate a clock C_{Cmp} with start time 0 and period P/n, i.e. $C_{Cmp} = N/n * CBase$, and to each node (agent in the OASIS translation) A in Cmp we associate a clock $C_A = N * CBase + m$, where m is the depth of A in Cmp.

Finally, it is important to notice, that the communications between agents belonging to different components have to be adjusted in order to take into account appropriately shifted past values.

For the sake of conciseness, we omit here the implementation details of the PsyC wrapper generation.

4.2 Functional Equivalence

The use of the PsyC code generation scheme presented in the opening of this section ensures that the function behaviour of each generated agent is exactly the same as that of the corresponding node in the SCADE model. Indeed, the same C functions generated by the SCADE KCG code generator are used in both contexts. Therefore, in order to demonstrate the functional equivalence of the entire application, we only have to show that the data are produced and consumed by the OASIS agents in the same order as

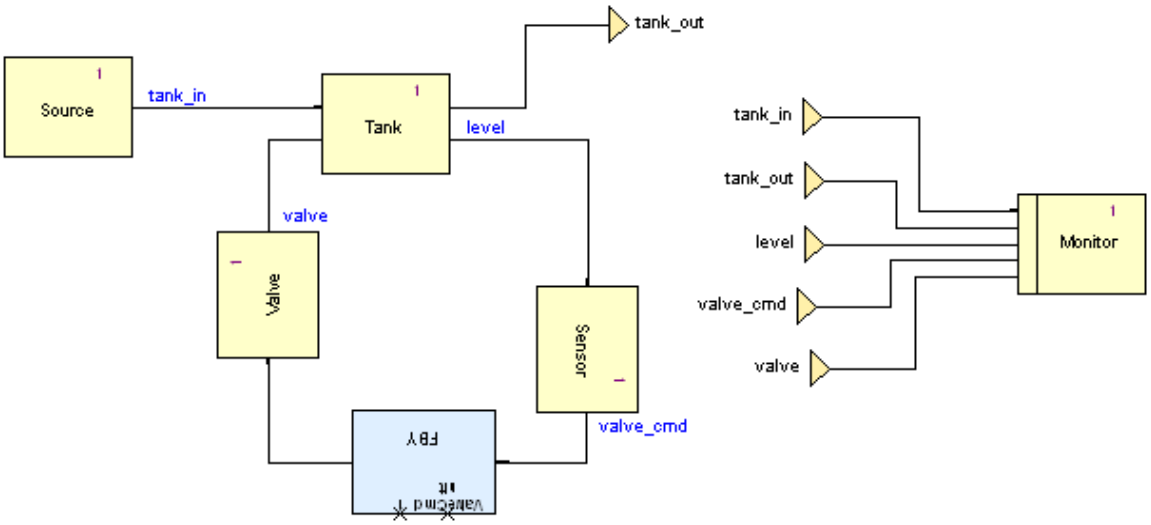


Figure 5. SCADE model for level control in a water tank.

in the source SCADE model. The argument below is can be illustrated by Fig. 6.

The order in which data are produced and consumed by SCADE nodes is defined by the underlying data dependency graph, whereas in OASIS it is determined by temporal synchronisation points (TSP), that is by the combined use of **advance** instructions and logical clocks.

The algorithm described in the previous section, defines the logical OASIS clocks in such a manner that, within each strongly connected component of the data causality graph of the application, the arrangement of execution windows guarantees that dependent agents are never executed in parallel. Hence, for each agent, there is at least one TSP separating its execution window from those of the agents producing the required data. Thus all past data referenced by the agent and produced within the same strongly connected component is available when necessary. In the previous section, we have omitted the detailed presentation of the use of the **advance** statements and offsets in the references to the past values of temporal variables. However, this use is straightforward.

Finally, a similar argument shows that the data are produced and consumed correctly among the strongly connected components of the data dependency graph. Indeed, the offset used for assigning temporal slots to strongly connected components are determined by their depth in the dependency DAG. Therefore, by construction any data used by an agent within a given strongly connected component is produced by another agent within a preceding temporal slot.

5 Case Study

The case study, considered in this paper, represents a medium voltage protection relay that we have previously developed using the OASIS approach, called OASISepam [10].

5.1 Software Architecture of OASISepam

In the domain of medium voltage, a protection relay is a device that detects and isolates faults in the electrical network. The sensor measures the current that flows on the network, and transmits information (voltage) that is proportional to the magnitude of this current to the relay. The relay digitalizes this information, applies signal processing algorithms, compares to the user settings (threshold, time delay, etc), and takes control decisions. The software part of OASISepam is therefore composed of three stages: 1) the acquisition stage, 2) the measurement stage and 3) the protection stage. All tasks within these stages are periodic tasks. Fig. 7 shows the overall software architecture of OASISepam in terms of agents and their interfaces.

Names associated to lines are names of variables exchanged between agents. Black circles indicate the agent that owns the variable, that is can produce new values for this variable, whereas other agents can only consult these values. The remainder of this section details this software architecture.

Acquisition stage. The goal of this stage is to build for other tasks new available data items. Data items are periodically available from an ASIC (Application Specific Integrated Circuit) through an SPI (Serial Peripheral Interface) line, every 555 μsec. This temporal constraint defines the sampling rate of the status of the power network. The periodicity of all other tasks is specified by Sepam developers from this base sampling rate using a multiplicative factor.

This stage is implemented by a single OASIS agent, called `AgARGA`. Data items (i.e. input currents) consist in an array of four values, named `DataBufferI`, representing the current on the monitored lines (I1, I2, I3 and I0 for the neutral). To specify the periodicity

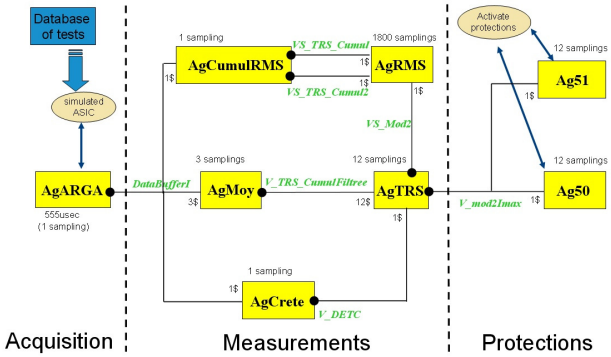


Figure 7. OASISepam: software architecture of the Sepam 10 using OASIS.

of the `AgARGA` agent, we define a clock called `HA_ARGA` with the granularity of 555 μsec. This clock serves as the reference clock for all other clocks in the system, that is other tasks refer to derived clocks built from `HA_ARGA`. A derived clock in OASIS is constructed using a multiplicative factor and an offset.

Measurement stage. The goal of this stage is to apply various algorithms in order to compute a value that will be used by the protection stage to first detect any power faults and then decide whether the safety-function of the protection relay should be activated. Therefore, the functionalities available within this stage depend on the protection algorithms embedded in the protection stage. In this paper, we assume that the measurement stage consists of: a computation of an average, a computation of the magnitude of the fundamental (50 Hz) and some harmonics (100 Hz, 150 Hz, etc.), a computation of a crest value and a computation of a root mean square. More details on the algorithmic part of this stage can be found in [13]. The average is computed whenever 3 new data items (i.e. input currents) are available. It is performed whenever 24 new data items are available and computes the last 12 averaged input currents (computed therefore using the last 36 data items).

The measurement stage is composed of five agents. The `AgCumulRMS` agent implements the sum and the square sum of 1800 data items, while the `AgRMS` agent implements the computation of the root mean square of the 4 currents. The temporal behaviour of the `AgRMS` agent is to be periodically activated every 1800 new data items, that is every 999 ms. Its associated clock `HA_RMS` is therefore defined in OASIS as 1800*`HA_ARGA`. The presence of the `AgCumulRMS` agent is explained by hardware memory constraints of the targeted board (64 Kb), and allows the `AgRMS` to only access a single data value—the last one provided by the `AgCumulRMS` agent. The `AgCrete` agent implements the computation of the crest value of all values of `DataBufferI`. Therefore, the temporal behaviour of this agent is to be periodically activated every 555 μsec, and relies on the same clock as `AgARGA`. This agent updates the temporal variable called `V_DETC`, of temporal rhythm `HA_ARGA`, at every activation. The `AgMoy` agent implements the computation of the average of the last 3 values of `DataBufferI`. Therefore, its temporal behaviour is to be periodically activated every 1.665 ms and its associated clock `HA_MOY` is defined as 3*`HA_ARGA`. This agent updates the temporal variable called `V_TRS_CumulFiltree`, of temporal rhythm `HA_ARGA`, at every activation. The `AgTRS` agent implements the computation of the magnitude of the fundamental and some harmonics of the inputs currents. The algorithms uses the last 12 values of `V_TRS_CumulFiltree`, which are computed using the last 36 available values of `DataBufferI` by `AgMoy`. In addition, the last available value of the `V_DETC` temporal variable is used. However, the temporal behaviour of this agent is to be periodically activated every 24 new data items, that is every 13.320 ms. Nevertheless, its associated clock `HA_TRS` is defined as 12*`HA_ARGA`. This is explained by the target end-to-end detection time of 26,640 ms [10]. This agent updates the temporal variable `V_mod2Imax`, of temporal rhythm `HA_TRS`, at every activation.

Protection stage. The goal of this stage is to embed the various protection algorithms that are required by a Sepam product. In this work, we considered that two protection algorithms were available: the 50 and 51 protections (protection codes are defined in [8]). Protection 50 is a protection against instantaneous phase over-current. It protects against the different possible phases short-circuits: three-phase or two-phase short-circuit. Protection 51 is a delayed 50 protection. Other types of protection are available in Sepam products from Schneider Electric.

Within OASISepam, the protection stage is made of two agents. Note that all protection agents follow the same temporal behaviour: they required to be activated every 13.320 ms. However, in order to achieve the target end-to-end detection time, we define the associated clock `HA_PROTECTION` as 12*`HA_ARGA` (i.e. 6.66 ms). As input, all agents rely on the last value of the temporal variable `V_mod2Imax`. Depending on the type of protection and its configured activation delay, protection agents ask (or not) the tripping of the circuit breaker. The `Ag50` agent (resp. `Ag51`) implements the 50 (resp. 51) protection.

5.2 Case Study implementation in SCADE

The SCADE implementation of the Sepam application is hierarchical (Fig. 8). The root node consists of the three sub-nodes corresponding to the application stages and a dummy sub-node. Indeed, one of the restrictions of the SCADE-OASIS transformation requires that the system be closed, i.e. all inputs and outputs must be connected, whereas the node TRS

produces a flow `VS_Mod2Imax`, not used in any of the protection elements of the case study and connected to the dummy node. This limitation can be levied in future versions. Presently, references to dummy nodes must be manually removed from the generated PsyC code (1 line of code per unused output).

The node **ARGA** models directly the acquisition functionality corresponding to the `AgARGA` agent of the OASIS version of the application. This node serves as an interface to the sensors (through the SCADE *sensor* feature) and provides the input values for the entire application. In the case study application, the values provided by the sensors are defined statically for testing purposes. The node **Protections** has two sub-nodes corresponding to the protection mechanisms described in the previous section. Finally, the node **Measurements** (Fig. 9) has five sub-nodes responsible for the collection of statistics.

Below, we illustrate some SCADE modelling elements providing a structured approach to the modelling of temporal behaviour of agents and communication among them. This approach can be easily generalised and systematically used in SCADE models intended for execution on the OASIS platform. The SCADE-OASIS transformation can be extended in order to identify these elements and generate appropriate native PsyC clocks and interface specifications (cf. Sect. 3.3). It is important to notice, however, that, although their systematic use would result in a more efficient PsyC code, it is not required. The transformation extension in question can be realised without compromising the backward compatibility.

Among the five sub-nodes composing the **Measurements** node, two—`CumulRMS` and `Crest`—must operate at the same rhythm as the `ARGA` node, which provides the input data. The other three nodes require additional clock constraints, since they operate with a slower rate (cf. Fig. 7). In SCADE, such constraints are implemented by introducing explicit nodes to generate a boolean signal representing the clock and using it to trigger a *Boolean activation block*. This results in additional computation in the generated C code. Since OASIS is based on a time-triggered approach, clock constraints do not require additional code and are handled directly by the system layer. Below, we present on the example of `AgentAverage` node the *design pattern* used in our case-study SCADE model (Fig. 10) that will allow future versions of the SCADE-OASIS transformation to exploit this feature of the OASIS framework.

Recall that the `AgentAverage` node must only be activated once in every three cycles and accesses the last three input samples to compute their average (cf. again Fig. 7).[1] In order to impose the activation rate, we use a SCADE Boolean activation block coupled with a dedicated node `Clocks::CA_AVG`, computing the corresponding SCADE clock by producing a tick on every third cycle. The Boolean activation block encapsulates a sub-node `Average` modelling the functional behaviour of the agent. We use an **fby** operator to ensure that the last value computed by this sub-node is maintained between consequent activations.

OASIS agents can refer to several past values of a given data flow (cf Sect. 3.3). Buffers necessary to hold these past values are managed by the OASIS system layer and their sizes are statically computed at compilation. In SCADE, these buffers must be modelled explicitly by ad-hoc use of **fby** operators. The design pattern described below allows systematic modelling of such buffers.

As described in the previous section, each sample provided by the Acquisition stage consists of four values representing the current on the monitored lines. In this model we have implemented separate buffering for each line. We use a combination of a **mapfold** with a one step memory shown in Fig. 11 to create a vector with the required sample values on a given line. The `D_TRS_NB_SAM_FILTER` parameter of the **mapfold** operator gives the number of samples that must be kept in the buffer (three in this case study). The **map** operator shown in Fig. 10 applies this construction to each line, with `D_ACQ_NB_LINES` parameter providing the number of lines to monitor (four in this case study).

5.3 SCADE-OASIS transformation

We have applied the current version of the SCADE-OASIS transformation to the SCADE model of the case study application described in the previous section. First of all, this has allowed us to test preservation of the functional semantics as stated in Sect. 4.2. Indeed, both Scade and OASIS semantics are deterministic and therefore systematically generate the same output flows provided the same input flows. We have observed identical values of the output flows generated by simulations in both environments, substantiating our claim of semantic equivalence. We also verified that the specification of the real-time constrained associated to the generated PsyC code is equivalent to the initial OASISepam application. Therefore, the end-to-end detection time requirements for OASISepam are also satisfied when the application is modeled in SCADE and then transformed into a PsyC code for the OASIS safety-oriented hard real-time tool chain.

6 Conclusion

Developing embedded safety critical real-time systems and ensuring properties such as deterministic

[1] Although for the `AgentAverage` node, the operation rate (once every three cycles) and the buffer size (access to the last three values of the input) coincide, this is not necessarily so in the general case.

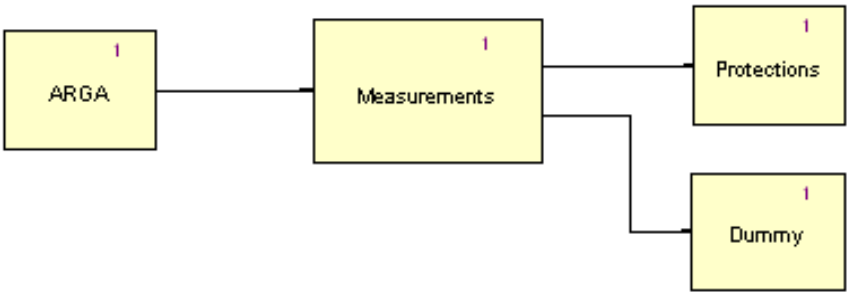


Figure 8. Root node of the SCADE model of the Sepam application.

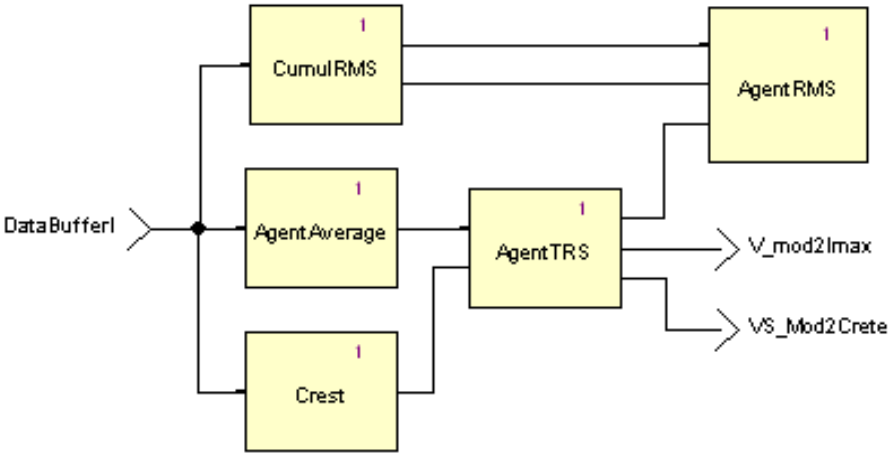


Figure 9. Measurements **node of the SCADE model of the Sepam application.**

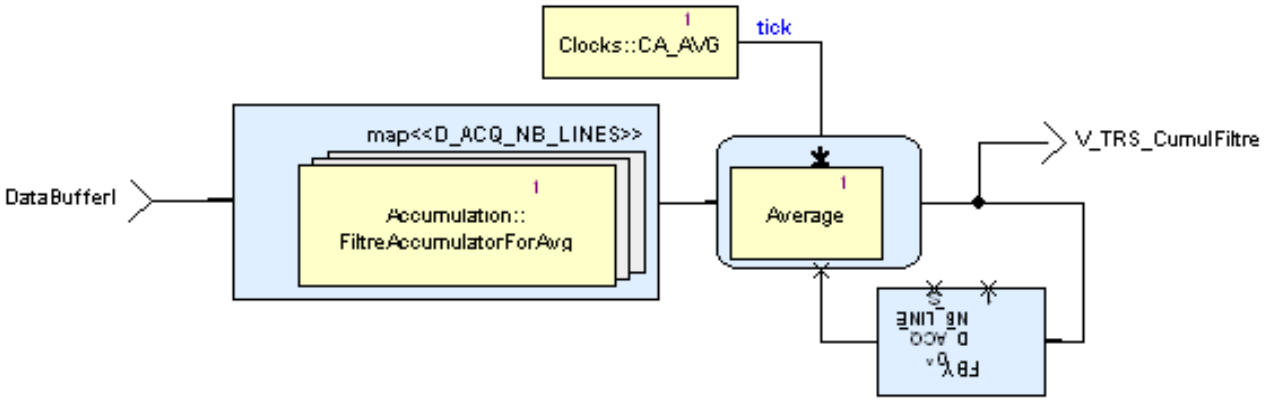
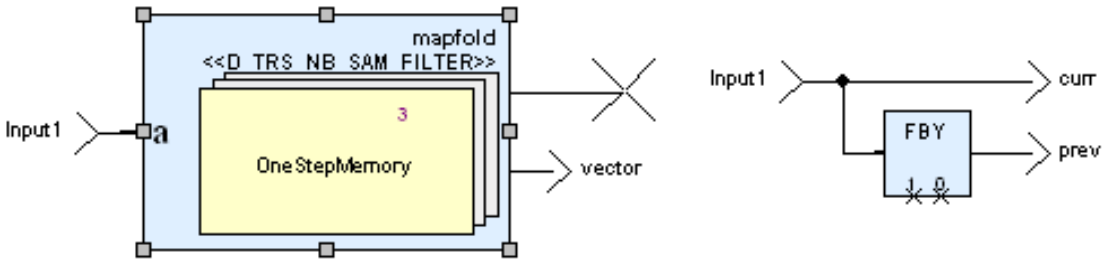


Figure 10. AgentAverage **node of the SCADE model of the Sepam application.**



(*a*) (*b*)

Figure 11. Buffer realised by a composition of a mapfold **(a) operator with one step memory (b).**

behaviour in a simple way for the application designers is a challenging task. In this paper, we have presented the OASIS safety-oriented real-time execution framework and an overview of the automatic transformation from the SCADE synchronous language models into applications for OASIS. In particular, we have presented a transformation preserving the functional semantics of the applications through an optimised arrangement of OASIS logical clocks.

We have also presented an industrial case-study that we have used to validate a partial implementation of the presented SCADE-OASIS transformation by comparing the trace of the simulation in SCADE Suite with that of the generated OASIS application. Temporal behaviour of the generated application model is equivalent to that of the application manually designed for OASIS.

Through the design of the SCADE model of the case-study application, we have exhibited a SCADE design pattern—also succinctly presented in the paper (Fig. 10 and Fig. 11)—that will allow future versions of the SCADE-OASIS transformation to exploit the specific advantages of the OASIS platform, namely optimised automatic sizing of communication buffers and multi-rate temporal behaviour. It should be noted, however, that, although using this design pattern would lead to a better optimisation of the generated OASIS application, already the existing implementation of the transformation can handle arbitrary SCADE models. In our future work, we will study a SCADE design pattern for the OASIS message box communication mechanism so as to allow arbitrary OASIS applications to be modelled in SCADE.

The OASIS approach can be transparently extended to distributed systems [9]. Future work will leverage this extension. Finally, we are currently working on a automatic transformation from the Modelica environment.

References

[1] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proc. of the 2003 ACM SIGPLAN conf. on Language, compiler, and tool for embedded systems (LCTES '03)*, pages 153–162, California, USA, 2003. ACM.

[2] D. Chabrol, V. David, C. Aussaguès, S. Louise, and F. Daumas. Deterministic Distributed Safety-Critical Real-Time Systems within the OASIS Approach. In *Proc. of the Int. Conf. on Parallel and Distributed Computing Systems (PDCS 2005)*, pages 260–268, Phoenix, AZ, USA, Nov. 2005.

[3] V. David, C. Aussaguès, S. Louise, P. Hilsenkopf, B. Ortolo, and C. Hessler. The OASIS Based Qualified Display System. In *4th American Nuclear Society Int. Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT)*, Ohio, USA, 2004.

[4] V. David, J. Delcoigne, E. Leret, A. Ourghanlian, P. Hilsenkopf, and P. Paris. Safety properties ensured by the OASIS model for safety critical real time systems. In *Proc. of the 17th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'98)*, volume 1516 of *Lecture Notes in Computer Science*, pages 45–59, Germany, 1998. Springer.

[5] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Proceedings of the Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*, Barcelona, Spain, Nov. 2008.

[6] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. Implementing multi-periodic critical systems: from design to code generation. In M. L. Bujorianu and M. Fisher, editors, *FMA*, pages 34–48, 2009.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, Sept. 1991.

[8] IEEE. Standard for electrical power system device function numbers, acronyms, and contact designations (c37.2-2008).

[9] M. Jan, J.-S. Camier, and V. David. Scheduling safety-critical real-time bus accesses using time-constrained automata. In *Submitted to the 19th Intl. Conf. on Real-Time and Network Systems*, 2011.

[10] M. Jan, V. David, J. Lalande, and M. Pitel. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In *Proc. of the 5^{th} Intl. Symp. on Industrial Embedded Systems (SIES 2010)*, pages 128–135, Univ. of Trento, 2010.

[11] C. M. Kirsch, M. A. A. Sanvido, T. A. Henzinger, and W. Pree. A giotto-based helicopter control system. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 46–60, London, UK, 2002. Springer-Verlag.

[12] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, Jan. 2003.

[13] K. Rahmouni, P. Gerin, S. Chabanet, P. Pianu, and F. Pétrot. Modelling and architecture exploration of a medium voltage protection device. In *Proc. of the IEEE Fourth Intl. Symposium on Industrial Embedded Systems (SIES 2009)*, pages 46–49, Lausanne, Switzerland, July 2009.

[14] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, 2008.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.