Component-based analysis of embedded control applications

Christo Angelov · Wei Guan · Nicolae Marian · Feng Zhou · Krzysztof Sierszecki · Søren Top

Abstract The widespread use of embedded systems requires the creation of industrial software technology that will make it possible to engineer systems being correct by construction. That can be achieved through the use of validated (trusted) components, verification of design models, and automatic configuration of applications from validated design models and trusted components. This design philosophy has been instrumental for developing COMDES—a component-based framework for distributed embedded control systems. A COMDES application is conceived as a network of embedded actors that are configured from instances of reusable, executable components—function blocks (FBs). System actors operate in accordance with a timed multitasking model of computation, whereby I/O signals are exchanged with the controlled plant at precisely specified time instants, resulting in the elimination of I/O jitter. The paper presents an analysis technique that can be used to validate COMDES design models in SIMULINK. It is based on a transformation of the COMDES design model into a SIMULINK analysis model, which preserves the functional and timing behaviour of the application. This technique has been employed to develop a feasible (light-weight) analysis method based on runtime observers. The latter are conceived as special-purpose actors running in parallel with the application actors, while checking system properties specified in Linear Temporal Logic. Observers are configured from reusable FBs that can be exported to SIMULINK in the same way as application components, making it possible to analyze system properties via simulation. The discussion is illustrated with an industrial case study—a Medical Ventilator Control System, which has been used to validate the developed design and analysis methods.

Keywords Embedded control systems · Component-based design · Domain-specific frameworks · Model-based analysis · Semantics-preserving model transformation · Runtime observers

1 Introduction

The widespread use of embedded systems poses a serious challenge for software developers who have to address a number of stringent and contradictory requirements: reduced development costs and time to market, error-free operation and predictable behaviour under hard real-time constraints, open architecture featuring reusable components and software reconfiguration, etc.

The conventional development process cannot easily cope with these problems, since it is largely based on informal design methods and manual coding techniques. This has a negative impact on both the economy of production and the safety of embedded systems. In particular, software safety is severely affected by design errors that are typical for informal
design methods, as well as implementation errors that are introduced during the process of manual coding.

Therefore, it is necessary to develop new design methods that will make it possible to engineer systems that are correct by construction. This is an ambitious goal that can be eventually accomplished by combining two complementary methodologies, i.e. model-driven and component-based design of embedded software [1]. The main elements of this approach include:

- Repositories of prefabricated and validated (trusted) components that can be used to build applications in a particular application domain,
- Computer-aided design of applications using formal design models that are appropriate for the application domain,
- Validation of design models with respect to functional and non-functional requirements, using feasible analysis methods based on semantics-preserving transformation of design models into appropriate analysis models,
- Automatic configuration of applications from validated design models using prefabricated software components.

It can be expected that the adoption of the outlined approach will result in the creation of industrial software technology for embedded applications, similar to those already available in mature areas of engineering, such as electronic design, mechanical engineering, etc.

The above considerations have been instrumental in developing the COMDES framework [2]. This is a software framework for time-critical distributed control applications, featuring a hierarchical component model and signal-based communication between components at all levels of specification. With this framework, an application is composed from actors, which are configured from prefabricated FBs. This is an intuitive and simple model that is easy to use and understand by application experts, i.e. control engineers.

The validation of design models is an important aspect of the overall development process. There are a number of analysis methods and tools developed over the years, which are now widely used by the engineering community. Therefore, our approach has been to use such tools rather than to invent new ones. However, that is possible if the analysis models used as input to those tools are consistent with the design models. This requires a semantics-preserving transformation of COMDES design models into analysis models used by the tools under consideration. Such transformations have already been developed for two tools that are widely used by the engineering community, i.e. UPPAAL¹ and SIMULINK², the latter being a de facto standard development environment for control and system engineers.

This paper presents the transformation of COMDES design models into consistent SIMULINK models. That is accomplished by wrapping components into S-FBs [3], which are then used to build the corresponding analysis models, taking into account the functional and timing aspects of system behaviour. This makes it possible to export design models to the SIMULINK environment and investigate system behaviour via simulation.

The above technique has been further extended to incorporate the verification of formally specified correctness properties by means of runtime observers. The latter are implemented using reconfigurable COMDES components that can be exported to SIMULINK using S-FBs in the same way as application components, making it possible to investigate system properties via simulation. The execution of runtime observers in SIMULINK provides a feasible analysis method, which can be used when other methods fail because of computational complexity.

The discussion is illustrated with a running example—a Medical Ventilator Control System [4], which has been implemented in COMDES and validated in SIMULINK. The rest of the paper is structured as follows: Sect. 2 presents an overview of the COMDES framework, focusing on the main features of its design models and the related software development process. Section 3 presents the transformation of COMDES design models into SIMULINK models that preserve the functional and timing behaviour of the original models. Section 4 presents component-based models of runtime observers used to monitor system properties, and their integration into the respective design and analysis models. Section 5 presents related research, and Sect. 6 concludes the paper by presenting a summary of the proposed analysis method.

2 Comdes framework: an overview

2.1 COMDES design models

COMDES is a domain-specific framework, which combines open system architecture with a model of computation that guarantees highly predictable behaviour, in the context of hard real-time distributed control systems [2, 5]. Its main features are summarized below:

A complex control system is decomposed into functional subsystems. A subsystem consists of one or more actors, i.e. active objects that are considered to be units of functionality as well as units of concurrency, such as sensor, controller, actuator, etc. A distributed embedded application is modeled as an actor network. Actors interact by exchanging labeled messages (signals), carrying information about state vari-


Springer
ables, such as pressure, temperature, etc. Signal-based communication is characterized as asynchronous (non-intrusive) multicast communication, which is transparent, i.e. independent of actor allocation.

The COMDES application model is illustrated with Fig. 1, which shows the actor network specifying the Medical Ventilator control system implemented in the case study [6]. It consists of five communicating actors that can be grouped into two subsystems—Ventilation Control and MMI Communication. The first subsystem consists of actors Sensor, Controller and PhaseSwitch. The Sensor actor reads signals from A/D converters and calculates the values of process variables such as pressure and airflow which are used as input data by the Controller actor. The latter implements a modal control system with several modes of operation, whereby the transitions between various modes are triggered by signals generated by the PhaseSwitch actor. The second subsystem consists of the Transmitter and Receiver actors, which are used to maintain communication with a Man–Machine Interface (MMI) unit over a serial link. System actors communicate with one another by means of labeled messages whose identifiers are shown on top of the corresponding communication links (see Fig. 1).

An actor is modeled as an integrated circuit consisting of a signal processing unit (SPU) and I/O latches, which are composed of input and output signal drivers, respectively (see e.g. Fig. 2 showing the Controller actor of the above case study). The input latch is used to receive incoming signals and decompose them into local variables that are processed by the SPU. The output latch is used to compose outgoing signals from local variables produced by the SPU and broadcast them to potential receivers. Physical I/O signals are treated in the same manner but in this case, the latches invoke hardware-specific routines in order to exchange physical signals with the environment.

The SPU is modeled as an acyclic FB network, configured from instances of prefabricated components—FBs. These are reusable and reconfigurable components that are stored in a repository in executable, binary format. FBs can be used to engineer heterogeneous embedded applications, such as sequential, continuous and hybrid (modal) control systems, or any combination thereof. The framework defines several kinds of FB—basic, composite, signal generator (SG) and state machine (SM) FBs.

Basic and composite FBs are components implementing various signal processing and control functions, e.g. linearization, scaling, limit/gradient checking, filtering and control algorithms, etc.

The SM FB implements the reactive (control flow) aspect of actor behaviour by indicating the current control action to be executed, in response to a particular event. The SG implements the transformational (data flow) aspect of actor behaviour. It is a composite component encapsulating alternative sequences of FBs, used to execute various control actions indicated by a master SM.

The SM and SG FBs can be composed together to implement actors with stateful behaviour, e.g. those used in sequential control systems as well as hybrid (modal) control systems, such as the Controller actor of the Medical Ventilator case study (see Fig. 3). Its signal processing unit contains an SM FB instance implementing the state transition graph shown in the figure, where each state (mode) is associated with a particular control action. Control actions are executed by the SG, which encapsulates instances of FBs PID and 2Multiplexor, whose functions are invoked in order to execute the indicated control actions.
Actors are executed in accordance with a *clocked synchronous* model of computation [7] known as distributed timed multitasking (DTM), which can be used to engineer highly predictable real-time systems that are free from input and output jitter and provide for a constant delay from sampling to actuation [5]. In accordance with this model, a control actor is mapped onto a real-time task having three parts: *task input*, *task body* and *task output*, implementing the input latch, SPU and output latch, respectively. One or more tasks may be allocated onto a particular processor, and their execution is managed by a real-time kernel, supporting split-phase execution of real-time tasks.

*Split-phase execution* of actor tasks is a characteristic feature of DTM. With this model of computation, the task body is executed in separation from task inputs/outputs. The latter are short pieces of code that are executed atomically in logically zero time, which is a valid assumption for embedded control applications. The task body is executed in a preemptive priority-driven scheduling environment and has a non-zero response time (see Fig. 4).

Task inputs and outputs are invoked at precisely specified time instants. In particular, the task input is executed when the task body is released, and task output—when its deadline arrives or immediately after the computation is finished if no deadline has been specified (see Fig. 4). Consequently, task I/O jitter is effectively eliminated as long as the task body is schedulable, i.e. its response time is less than the task deadline.

That mode of operation can be used with both periodic and aperiodic real-time tasks. It can also be extended to task sequences implementing phase-aligned distributed transactions. In that case, I/O signals are generated at transaction release/deadline instants, thereby eliminating transaction I/O jitter [5].

The DTM model of computation is presently supported by the timed multitasking version of the HARTEX kernel, which provides an operational environment for COMDES-based embedded applications [8].

### 2.2 Software development process

The COMDES software development process is aimed at eliminating both design and implementation errors, which will hopefully result in software that is *correct by construction*. That is to be achieved through the use of validated (trusted) components, verification of design models and automatic configuration of applications from validated design models and trusted components. On the other hand, timed multitasking makes it possible to engineer highly predictable systems operating in a flexible, dynamic scheduling environment.

The use of prefabricated *executable* components is an important feature of the COMDES software development process. Consequently, an application is *configured* from instances of prefabricated components using a validated design model, rather than generating the entire application code from the design model.

The COMDES development process has three different aspects, i.e. component repository development, application configuration and application validation. These aspects are supported by the corresponding subsystems of the COMDES toolset [6,9], i.e. *Component Development Toolset*,
Application Configuration Toolset and Application Analysis Toolset (see Fig. 5).

The first subsystem supports computer-aided development of component repositories, including artefacts such as component source codes, executable codes and test cases for various platforms, component S-functions, etc. The second subsystem incorporates a graphical application editor used to configure an embedded application from instances of prefabricated and validated software components. Application design models are validated in the third subsystem, which consists of model transformation tools as well as associated analysis tools, such as SIMULINK, UPPAAL and eventually MAST.3

The validated design models are ultimately used to configure the application from component executables that are fetched from the repository, as well as glue code generated in the process of configuration. During that process, the COMDES actor model is transformed into a HARTEX task model, which is then used to automatically generate the embedded application from instances of prefabricated components.

The COMDES development process is facilitated by the principle of separation of concerns, which is an important feature of the COMDES framework. This makes it possible to separately treat different aspects of complex systems, such as system structure and behaviour, computation and communication, functional and timing behaviour, reactive and transformational behaviour, etc. [5].

Separation of concerns facilitates both the design and analysis of embedded systems, which is reflected in the adopted software development process and the supporting software engineering environment. Consequently, different aspects of system behaviour can be analyzed in separation using appropriate techniques and tools, following a semantics-preserving transformation of system design models into the corresponding analysis models.

In particular, the behaviour of predominantly continuous systems can be analyzed through simulation by exporting design models to SIMULINK, using numerical techniques that have been widely accepted by the Control Engineering community. Simulation can also be used with discontinuous (sequential) control applications, which are usually modelled as systems of interacting state machines. However, such systems are often characterized by complex behaviour that cannot be easily and exhaustively analyzed through simulation and testing. In this case, system behaviour must be analyzed through formal property verification, using model-checking tools such as e.g. UPPAAL. Unfortunately, model checking is often hampered by computational complexity. This problem can be partly overcome by developing light-weight analysis methods based on the concept of run-time observers, which can also be exported to SIMULINK, making it possible to check system properties via simulation.

Finally, separation of concerns makes it possible to analyze timing behaviour in separation from functional behaviour, by means of numerical response-time analysis techniques and tools, such as e.g. MAST.

3 Semantics-preserving Comdes-Simulink transformation

The main idea of our approach is to export the original design model of the system under investigation to the SIMULINK environment and analyze it via simulation, such that the original execution semantics is preserved during the simulation. This transformation is facilitated by the similarity between COMDES design models and SIMULINK analysis models representing the control system, both of which are data flow models.

COMDES employs a hierarchical data flow model [2,5] whose main features have been presented in the preceding section. At application level, a control system is specified as a network of actors interconnected by communication channels that are used to exchange signals. System operation is specified in terms of distributed transactions involving one or more actors whose execution order is subject to precedence constraints that are derived from the flow of signals between the actors (e.g. actors Sensor and Controller in Fig. 1).

At the actor level, an actor is modeled as an acyclic network of FBs interconnected by signal lines that represent the

3 MAST: http://mast.unican.es/.
data flow within the actor (see Figs. 2 and 3). A FB network can be referred to the class of synchronous data flow networks, whose execution is controlled by a static schedule that is derived from the flow of signals—from inputs to outputs [2].

It has been shown that the behaviour of a COMDES control system can be formally specified in terms of functions defining signal transformations—from input signals to output signals, taking into account constant delays introduced by transaction deadlines [5]. It is assumed that the transaction is executed periodically, being triggered at discrete time instants $kT$, $k = 1, 2, \ldots$, such that $y(kT + D) = F(x(kT))$, where $F$ is a composite function specifying signal transformations—from input signals $x$ to output signal $y$, $T$ is the transaction period, and $D$ is the transaction deadline, $D \leq T$.

A SIMULINK model is very similar, being composed of blocks and signal lines [3]. Constituent blocks are either standard SIMULINK blocks or S-function blocks (S-functions, for short) that are supplied by the user in order to import external, user-developed code into the SIMULINK model [10]. This is once again a data flow model defining a static execution order for the constituent blocks. Hierarchy can be modeled using SIMULINK subsystems. In particular, subsystems are used to model composite components as well as application subsystems, such as actors.

The SIMULINK model is executed in a synchronous fashion, assuming a zero delay from inputs to outputs. In particular, the control system blocks are executed at discrete time instants $kT$, $k = 1, 2, \ldots$, such that $y(kT) = F^*(x(kT))$, where $F^*$ is a composite function specifying signal transformation—from inputs to outputs of the SIMULINK diagram modeling the control system.

There is an obvious similarity between the two models, which facilitates model transformation. There are however a number of issues that have to be addressed, so as to take into account the functional and timing aspects of system behaviour and ultimately, develop an analysis model that operates in exactly the same manner as the original design model. In particular, it is necessary to configure the modelling function $F^*$, such that $F^* = F$, and at the same time—introduce the constant input/output delay $D$, which is an essential feature of the COMDES model of computation (see following sections).

3.1 Transformation of functional behaviour

The similarity of COMDES and SIMULINK models makes it possible to export a COMDES design model to the SIMULINK environment, by wrapping COMDES components into S-functions and wiring them together, following the interconnection pattern of the original design model. This kind of transformation can be characterized as heterogeneous two-plane modeling in SIMULINK (see Fig. 6).

With this modeling technique each FB of the original COMDES model is wrapped into an S-function. S-functions operate in the SIMULINK plane and are interconnected with each other using ports.

When activated, S-functions invoke the encapsulated FBs, which operate in the COMDES plane. These communicate directly with each other in accordance with the COMDES softwiring technique [2], whereby a FB uses pointers to access the output buffers of other FBs to fetch the necessary input data.

In order to implement this modeling technique, it is necessary to create a library of wrapped COMDES components in the form of S-functions. It can be easily seen that the wrapped components library is a one-to-one mapping from the original COMDES library (see Fig. 7).

It is also necessary to figure out a way of mapping the COMDES softwiring technique to the Simulink interconnection technique. Finally, it is necessary to find appropriate modeling techniques for complex components such as composite and SG FBs.
3.1.1 Transformation of basic and state machine function blocks

Each S-FB from the wrapped component library encapsulates the corresponding COMDES FB, e.g. Comparator, Counter, etc. The S-FB communicates with other components in the Simulink plane via input and output ports, whereas an FB uses input pointers to access memory locations containing input data. The original version of the wrapping technique is shown in Fig. 8. It assumes that the encapsulated FB has access to S-function input ports via the corresponding input pointers (shown as dashed arrows), and the data stored in the FB’s output buffers is copied to the output ports of the S-function.

The connection between two S-functions say SF_A and SF_B is shown in Fig. 9. In this case SIMULINK takes charge of copying the data from output ports of SF_A to the input ports of SF_B.

Unfortunately, the above technique does not preserve the FB interaction mechanism used in the original design model [2], i.e. the use of pointers to access the output buffers of producer FBs from within the consumer FB.

This problem can be avoided by copying the address of the FB output buffer to the corresponding output port of the S-function, instead of the output data itself. So, the buffer address will be transferred from the SF_A output port to the connected SF_B input port and finally assigned to the corresponding FB input pointer. The latter can be used to directly access the output buffer of the FB encapsulated in SF_A (see Fig. 10).

In this way, wrapped FBs communicate in exactly the same manner as in the original design model, whereby S-functions provide a shell with which the internal FB can be executed in the SIMULINK environment, and also—provide access points for monitoring signals in SIMULINK.

In COMDES, each FB is a type, which can have one or more instances, and each of them will execute a specific function (method) on the instance data (see e.g. function blocks FB_A and FB_B in Fig. 10). In SIMULINK, FB instances can be specified by means of the corresponding S-function mask. The mask is used to take the input parameters supplied by the user and pass them to the internal FBs. This configuration approach has been applied to all basic FBs in the wrapped component repository.

However, the above approach cannot be used when it comes to configuring instances of the State Machine FB, because in this case, the number of inputs and their data types vary with different instances, and each instance requires a different configuration structure (State Machine Table), containing the state transition graph of the implemented state machine.

To solve this problem, a dynamic link library (DLL) is used in conjunction with the S-function encapsulating the SM instance. With this approach, the State Machine Table is compiled to a DLL independently, and it can be subsequently used by the state machine S-function to implement the desired state machine behaviour. In this case, the S-function contains only the standard method of the State Machine type—the so called state machine driver, which is used to process the state machine table of the SM instance. Thus, the SM component can be wrapped into an S-function and used in different applications.

Furthermore, the input configuration of the state machine is also compiled into the DLL, so as to specify the number and type of inputs used by a specific SM instance, and their connections. The other configuration parameters (e.g. instance number, instance function, etc.) are left for the S-function mask, which is thus identical for all FB types.

3.1.2 Transformation of composite and signal generator function blocks

Composite FBs and SGs cannot be wrapped into S-functions in the same way as basic FBs. The reason is that they are hierarchical models, whereas the S-function is a flat model, which rules out the nesting of S-functions. However, that problem can be easily solved by means of SIMULINK subsystems encapsulating S-functions wrapping constituent COMDES components. A composite FB contains a single sequence of FB instances. Hence, it can be modeled by a single subsystem
A SG is composed of multiple FB sequences, which are selected for execution by a master state machine indicating the sequence to be executed during a particular invocation (see Sect. 2.1). In SIMULINK, each of these sequences is modeled by a component denoted as Switch Case Action Subsystem. These subsystems are triggered by a Switch Case block, as shown in Fig. 11, which depicts the SIMULINK model of the Controller SG from Fig. 3.

3.2 Modeling actor timing behaviour under timed multitasking

Timed multitasking is simulated by means of SIMULINK subsystems modeling the input and output latches of the actor (Fig. 2), as shown in Fig. 12.

Inside the Input Latch subsystem, the incoming messages are unpacked if they have more than one constituent variable. That is modeled by Demultiplexor (Demux) components, whose outputs are connected to zero-order hold (ZOH) blocks. These are used to sample input signals and keep them unchanged during the execution period of the actor task.

Inside the Output Latch subsystem, several output variables could be packed into one message. That is modeled by Multiplexor (Mux) components, whose outputs are connected to Integer Delay (ID) elements modeling the constant delay from sampling to actuation, as specified by the actor deadline.

The I/O latch subsystems are combined with a subsystem modeling the actor task, in order to compose a subsystem modeling a COMDES actor (see Fig. 13). The actor task is configured from connected S-functions which are chosen from the wrapped COMDES component library (see e.g. the Controller model, shown in Fig. 14). During simulation, the above subsystems have to be executed in a sequence modeling the split-phase execution of actor tasks: the actor task is released by the corresponding execution trigger, i.e. periodic timing event, whereby the input latch is executed when the task is released and the output latch—when the task deadline arrives, as shown in Fig. 4.

To that end, actor subsystems have to be appropriately parameterized. The ZOH blocks of the Input Latch keep the input signals of the actor task unchanged during the execution period, so that the sample time for these blocks should be equal to the actor period expressed as an integer number of simulation time units. The ID blocks in the Output Latch delay the output signals for an interval of time equal to the actor deadline, specified by the corresponding number of simulation time units.

Actor period and deadline parameters are supplied via a mask associated with the actor subsystem, e.g. Fig. 13, where the mask specifies an actor period of 6 ms and deadline of 2 ms. These settings are passed to the internal I/O latches.
Component-based analysis of embedded control applications

3.3 Building a SIMULINK model with the wrapped COMDES components

The outlined modeling technique can be used to build the models of the actors constituting the entire SIMULINK model under investigation, e.g. the Controller model shown in Fig. 14. These can be used to compose the system model as shown in Fig. 15. Here, the constants on the left-hand side are used to simulate the Receiver actor task’s output variables, which are combined into one message (i.e. RxMsg) sent to both the PhaseSwitch and the Controller actors, following the original design model (Fig. 1).

Besides that message, the two actors also receive and produce other messages/signals in the same way as in the design model, i.e. PSMsg1, PSMsg2, SMsg, etc. System actor models are connected to the Medical Ventilator plant model, which is derived from the real plant (inspiration valve in the Medical Ventilator).

The plant is coupled to the Controller actor via the pidControlValue control signal and SMsg.p1Flow feedback signal thus forming a closed-loop control system. Finally, the experimental result can be shown in the scope, which is connected to two copies of the plant model operating with and without control respectively, for the purpose of comparison. In this way, the control parameters can be tuned before they are used in the control software of the real machine, which saves both development time and cost.

Figure 16 shows the signals generated during the SIMULINK simulation run, i.e. the PhaseSwitch output signal—inexpExpFlag (the upper diagram) and the system step response (the lower diagram). The former indicates the different respiration phases (inspiration or expiration) to the Controller actor, which reacts accordingly by generating appropriate control signals for the inspiration and expiration valves of the machine.

The simulation result, i.e. the controlled step response (smooth curve), is shown together with the uncontrolled step response (oscillating curve) in the second diagram. The comparison between the two step responses shows that both oscillation and overshoot, which are dangerous to the patient, have been effectively eliminated by the implemented control system.

The presented analysis technique has been integrated into the COMDES Analysis Toolset. It is now possible to automatically generate S-functions from COMDES components, and automatically transform a COMDES design model into a consistent SIMULINK model via model-to-text transformation. The resulting textual description can also be represented in visual form, using the graphical interface of SIMULINK.

The next section extends the presented analysis method to runtime observers encoding formally specified correctness.
properties, which are exported to SIMULINK together with the application, and the corresponding properties are validated via simulation.

4 Runtime observers in Simulink

The analysis of design models is an important phase of the development process, which is aimed at validating the design before its actual implementation.

Formal property verification is indispensable whenever it is necessary to guarantee the functional correctness of embedded applications. Model checking is a verification method, which is highly popular in the Software Engineering Community and is presently supported by a considerable number of tools. Unfortunately, it is plagued by computational complexity, which is due to the phenomenon of state space explosion. That is why system engineers often prefer to use non-exhaustive yet feasible analysis methods, such as simulation. This is demonstrated by a considerable number of tools, e.g. Simulink [3], Giotto/TDL_in_Simulink [11], Ptolemy II [12], Metropolis [13], etc.

Another option is offered by light-weight analysis methods used to check program behaviour against formally specified properties, based on the concept of run-time observers (monitors) [14]. These methods are referred to as run-time checking or run-time verification of system properties [15]. As the name implies, run-time observers are executed during system operation. However, they can also be executed in the process of simulation, as is the case with property monitors used in Metropolis [13].

We have used a similar approach based on the concept of synchronous observer originally introduced in the context of synchronous programming languages [16,17]. Roughly speaking, an observer is a program encoding a specific correctness property, which might be stated formally. Synchronous observers are usually defined for safety properties. An observer is composed in parallel with the program it observes, whereby it emits a signal fail whenever the property is violated. Conversely, if the signal is never emitted, the application satisfies the property that is specified by the observer. Model checking can be used to guarantee that the signal fail cannot be emitted, making the proof automatic. This can also be done during real-time execution or simulation, using runtime observers. In general, an observer is needed for each requirement that has to be validated.

Synchronous observers have been originally conceived as programs that are implemented in the language of the monitored application program [16,17]. That is not the case with component-based design methods, which use higher-level languages, e.g. graphical notations describing an embedded application (or a specific subsystem) as a composition of component instances. With this approach, an observer may also be specified as a composition of component instances. A COMDES observer is an actor, which consists of a control block, called Temporal Evaluator (a finite state machine encoding a temporal operator), and a Predicate Evaluator, encoding the atomic proposition embedded in the temporal operator. These two blocks can be configured from instances of standard executable components—FBs (see next section).

Consequently, both application actors and property observers can be specified using the same modeling language, i.e. COMDES component/design models. Observer FBs can be transformed into S-FBs using the methodology presented in the preceding sections. Consequently, application and observer actors can be exported to SIMULINK, making it possible to verify the corresponding properties via simulation, using runtime observers.

Runtime (dynamic, on-line) observers monitor the execution of an implementation, checking that the execution trace satisfies a set of formalized temporal requirements specified in linear temporal logic (LTL) as an expression involving predicates that relate inputs and outputs with states at different time points [14,15]. The validation of the observer temporal property (temporal assertion) is carried out over a finite simulation sequence. The aim of this approach is to help find errors in reasonable time, while operating at a higher level of abstraction. Its application domain is usually associated with large systems where automated verification methods, such as exhaustive model checking, are not feasible due to memory shortage or timeouts.

Model-based runtime observers catch undesirable behaviors and property violations from the application model being verified. In order to check whether a controller C fulfills a particular property, a corresponding observer O is derived and added in parallel, forming a new system \( H(C \parallel O) \), as shown in Fig. 17. The verification problem is thus reduced.
to monitoring whether the resulting system $H$ activates the signal $fail$ in any state during simulation, signaling that the property has been violated.

A key requirement is that observing a controller does not modify its structure and behavior. A runtime observer must listen to signals generated in the controller and the plant in a fully transparent manner, observing the states of the system in the process of execution (simulation) and updating its internal state when the variables change value.

In the case of COMDES, this is guaranteed by the anonymous and non-intrusive nature of actor interactions, which are carried out using labeled messages (signals) and content-oriented message addressing. The observer is thus able to listen to the signals broadcast by application actors in a transparent and non-intrusive fashion, ruling out undesirable interference with the control system observed. Signal-based communication is readily modeled in SIMULINK, being inherent to its model of computation.

The advantages of the runtime observer techniques compared with model-checking are that both observer and controller can be implemented in the same language, and verification takes much less time compared to model checking. On the negative side, this technique does not guarantee exhaustive property verification over the entire state space (which may not be possible or at least—feasible). In that case, verification is limited to a finite execution trace over a limited time interval. Consequently, the use of run-time observers during simulation does not check conformance with a specific requirement, but only that the implementation does not violate the requirement during the simulation run.

4.1 Component-based design of runtime observers

The observer structure is based on the fact that property evaluation is a cyclic process over a finite sequence of states, whereby in each state the input data (signal inputs and other internal process variables) are checked and the generated truth values become obligations for the future, carried out by a temporal state machine. In the case of safety and liveness properties expressed in LTL logic, the formulae will be:

$$\text{Always} (f) = f \land \text{Always} (f)$$
$$\text{Eventually} (f) = f \lor \text{Eventually} (f),$$

where $f$ is a stateless predicate expression.

The actual observer implementation takes advantage of the component architecture of the COMDES framework, where stateful and stateless components are functionally well-defined and structurally separated, but composed together in order to realize complex functionality. Accordingly, an observer can be implemented as a composition of components, whereby the atomic predicate part of a temporal formula is evaluated by a stateless $\text{Predicate Evaluator}(PE)$, whereas temporal monitoring is performed by a finite state machine—the $\text{Temporal Evaluator}(TE)$ (see Fig. 18).

The PE and TE components are configured from instances of COMDES FBs. These are encapsulated into an observer actor, which may be triggered for execution by a periodic timing event or message arrival event. Its input signals are latched when the actor task is released and then passed onto the encapsulated FB instances, whose execution order is defined by the flow of signals—from inputs to outputs.

An example implementation of the safety property $Gp$ is presented in Fig. 19. Similar patterns have been developed for the liveness property $Fp$, as well for other properties, such as $p \rightarrow q$, $p \rightarrow Gq$, $p \rightarrow Xq$, etc.

Observer components can be wrapped into S-functions like application components and thus exported to SIMULINK. This makes it possible to analyze formally specified properties through simulation, using runtime observers.
4.2 Property monitoring via run-time observers

The presented analysis methodology has been tested using once again the Medical Ventilation Case Study. The latter is characterized by a number of temporally constrained functional requirements, e.g. the safety property: “The inspiration valve must be closed when the expiration valve is opened.”

Based on the $LTL$ formula patterns given in [18], the above property can be represented in the form:

$$G ((q \land F\neg q) \rightarrow (p \lor \neg q)) ,$$

where the predicates are specified as $p$:inspiration_valve IS CLOSED, and $q$:expiration_valve IS OPENED. The inspiration valve is closed when the control signal $pidControlValue$ is equal to zero, whereas the expiration valve is opened when the on/off control signal $expValveOutput$ is false. These two signals are generated by the Controller actor (see Fig. 3).

The above signals are processed by the run-time observer. It can be implemented using COMDES components, i.e. comparator, inverter, imply and AND FBs coupled to the corresponding temporal evaluator state machines (see Fig. 20).

The run-time observer is composed recursively following the exploration of the temporal formula—from levels 1 to 3. It is executed as a COMDES actor with data flow semantics, which defines the corresponding execution order of the FBs involved.

The observer has been exported to SIMULINK together with the application (see shaded box shown in Fig. 15). The experiments carried out have demonstrated that the above property is valid, with the observer generating an OK signal throughout the simulation run, without any noticeable overhead demonstrated during the process of simulation.

5 Related research

The presented analysis method is based on a semantics-preserving transformation of COMDES design models into SIMULINK models, which are then used to validate the design through simulation, or through property verification using runtime observers. The validated model is subsequently used to automatically configure the application from prefabricated executable components stored in a component repository.

That is contrast to the conventional design process, which employs manual coding or automatic code generation from validated models. The process usually starts with a control system design, followed by modeling and simulation in SIMULINK in order to validate the model, which is then used to generate the entire application code, using tools, such as the SIMULINK Real-Time Workshop by MathWorks or TargetLink developed by dSPACE GmbH.

The code generated from SIMULINK models is encapsulated into periodic tasks running under some kind of real-time kernel. Such code is usually appropriate for rapid prototyping but it is not always optimal for hard real-time embedded applications. For example, the conventional implementation does not separate task input/output and communication from task execution, resulting in variable task response time, and consequently—variable delay from sampling to actuation and I/O jitter.

This problem has been addressed in systems that employ the concept of Logical Execution Time (LET) programming, like Timed Multitasking [12], Giotto [19], xGiotto [20], as well as COMDES [2]. LET programming is characterized by split-phase execution of real-time tasks, whereby task bodies are executed in separation from input and output drivers. The latter are executed atomically at precisely defined time instants (e.g. at the beginning and end of period), resulting in a constant delay from input to output, which is denoted as the task logical execution time.

The adoption of such a model requires that its semantics be preserved in the analysis models of tools used to validate the design, e.g. SIMULINK. This has motivated the development of appropriate model transformation techniques, e.g. SIMULINK modeling of Giotto/TDL applications [11]. One of the techniques discussed in that paper is based on the introduction of task blocks featuring Zero-Order Hold and Unit Delay elements, used to model the semantics of Giotto tasks, which are characterized with a one-period delay from input to output. We have adopted a similar approach using Zero-Order Hold to freeze input signals throughout actor execution but we use Integer Delay elements in order to model logical execution times of arbitrary duration (e.g. less than period), as required by the DTM model of computation. This can also be accomplished by using enabled subsystems modeling task input, task body and task output, respectively, similar to
another technique of Giotto task modeling discussed in [11]. However, that solution would complicate the overall system model since it requires the modeling of the timed multitasking kernel, in order to generate the enable signals for various task subsystems.

Another important feature of our approach is the use of runtime observers for checking system properties through simulation. The concept of observer has been originally introduced in the context of so-called synchronous languages [7]. Synchronous observers can be used to perform property verification via model checking, which is applied to the synchronous product of the corresponding state machines derived in the process of compilation. However, the observer can also be executed in real-time, thus becoming a separate module, i.e. a runtime observer monitoring the behaviour of the embedded application.

Unfortunately, in that case modularity is purely conceptual since it is compiled away in the process of program translation. This makes it difficult to apply synchronous observers in a simulation environment, in strict separation from the application program. On the other hand, the execution time of the observer is added to that of the application program, which may result in increased response time and complicated timing analysis.

The above problem is partially resolved in the Java implementation of run-time monitors [14], where the monitor is executed in separation from the monitored application program. However, the communication between the two programs requires that instrumentation code be inserted into the application program, in order to monitor particular state variables and related propositional variables (events), and transmit their values to the monitor whenever they are updated. However, this may change the original behaviour of the monitored application program.

In the proposed method, observers are completely decoupled from the application. That is due to signal-based communication, which makes it possible to execute observers in a transparent and non-intrusive fashion, which rules out undesirable influence on the functional and timing behaviour of the application.

Using the same language for both applications and observers is a feature shared by synchronous languages, Java monitors and COMDES. However, in our case observers are configured from standard FBs using a high-level graphical language, in the same way as application actors. This makes it possible to export observers to SIMULINK, together with the application, and monitor system properties via simulation.

The Metropolis framework also employs runtime observers (monitors) for checking properties through simulation [13]. However, in Metropolis, designers use logic of constraints (LOC) formulae to specify quantitative properties that are translated into simulation monitors in C++.

The monitors analyze execution traces and signal LOC formula violations. Like other simulation-based approaches, this makes it possible for a monitor to disprove a LOC formula if it finds a violation but it cannot prove its correctness conclusively, as this requires exhaustive analysis of execution traces. However, that is the technique of choice when formal verification fails because of memory and time limitations.

6 Conclusion

The paper presents an analysis technique for component-based embedded applications in the context of the COMDES framework, which is based on a semantics-preserving transformation of COMDES design models into SIMULINK analysis models.

A two-plane modeling approach to the analysis of embedded applications has been developed using wrapped COMDES components (S-functions). In the upper plane, the S-functions are chained and simulated in Matlab SIMULINK. The dataflow between the components is maintained the same as in the COMDES model, and the data themselves are actually processed by the encapsulated lower-plane COMDES components. This approach has been further extended to composite components and system actors.

The presented model transformation preserves the COMDES semantics in terms of functional behaviour, which remains unchanged in the SIMULINK model. The timing behaviour of COMDES applications is also preserved via specific solutions making it possible to sample input signals and produce output signals at precisely specified time instants, in accordance with the timed multitasking semantics of COMDES design models.

The above analysis technique has been further extended to incorporate the monitoring of formally specified correctness properties using runtime observers. This is a lightweight analysis method which can be used in case exhaustive verification is unfeasible because of computational complexity. In our case, the application and the observers are implemented in a uniform fashion using design models provided by the COMDES framework, whereby the observers are composed from reconfigurable COMDES components (FBs). Hence, observers can be exported to SIMULINK using S-functions, in the same way as application components, making it possible to monitor system properties via simulation.

The developed methodology is not strictly limited to COMDES applications. In a broader context, it offers an analysis method specifically tailored for embedded applications built from pre-fabricated executable components that are different from SIMULINK blocks. In that case, it is not possible to follow the conventional development process, whereby a control system is initially designed and simulated in SIMULINK, followed by code generation from the
validated system model. This problem is addressed by the presented methodology, which makes it possible to configure an application from prefabricated components, and then validate the design by exporting it to SIMULINK, such that the execution semantics of the original design model is preserved.

References