Hardware-in-the-Loop Simulation of Component-Based Embedded Systems
Guo, Yu; Zhou, Feng; Marian, Nicolae; Angelov, Christo K.

Published in: Proceedings of the 8th International Workshop on Research and Education in Mechatronics REM 2007

Publication date: 2007

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 10. Jan. 2017
Abstract—The paper presents COMDES-II - a software framework for distributed control systems. The framework provides for systematic top-down specification of distributed applications: a system is conceived as a composition of embedded actors that communicate transparently by exchanging labeled messages (signals), and actors are composed from reusable executable components (function blocks). The framework and its software components have been validated in a number of real-time control experiments, including the Production Cell Case Study. The paper presents the software design models of the Production Cell control system, based on COMDES-II, focusing on the system engineering aspects of the software design. The latter has been formally verified using the Uppaal verification tool. To that end, a method has been developed to transform the COMDES-II interaction diagram into an Uppaal model, using networks of automata representing system actors, as well as controlled objects from the Production Cell environment. The developed control system design has been tested via hardware-in-the-loop simulation involving a real-time control network and an animated computer model of the plant.

Index Terms—component-based design, software framework, actors, function blocks, hardware-in-the-loop simulation

I. INTRODUCTION

The widespread use of embedded systems mandates the development of industrial software design methods, featuring formal models (frameworks) and prefabricated software components, as well as computer-aided software engineering environments [1, 2]. This has motivated the development of the COMDES-II framework (Component-Based Design of Software for Distributed Embedded Systems, v. II) [3]. It defines a consistent set of executable models specifying relevant aspects of systems structure and behaviour within the domain of distributed embedded systems for mechatronic applications.

COMDES-II allows for systematic specification of reconfigurable and reusable components and embedded applications. This is done in a top-down fashion using a two-level system model. At the top level, a system is conceived as a composition of embedded actors that communicate transparently by exchanging labeled messages (signals) over a real-time network. At the second level, actors are modeled as reconfigurable active objects that are composed from executable components, such as basic, composite, state-machine and modal function blocks. The use of well defined and validated (trusted) components eliminates the errors caused by manual coding, which still dominates traditional embedded software development methods.

The COMDES-II software design method consists of three stages involving application modeling, application analysis (verification) and validation (testing). It has been experimentally investigated using the well known Production Cell Case Study [6]. The paper presents the software design of a distributed computer control system of the Production Cell based on the COMDES-II framework and its components. The behaviour of the system, and in particular, the communication between COMDES-II actors, is specified by means of an augmented message sequence chart (actor interaction diagram). During the design stage, a verification process is applied to that diagram in order to find possible errors in the specification as early as possible, using the Uppaal verification tool [5]. The paper presents a method to transform the COMDES-II message sequence chart into an Uppaal model via networks of automata, structured data types, and channel synchronization. The Uppaal model contains both automata models that are part of the control system, and models of the controlled objects, i.e. physical units from the Production Cell environment. The developed control system design has been eventually tested via hardware-in-the-loop simulation involving a real-time control network and an animated computer model of the plant running in a PC.

The rest of the paper is structured as follows: Section 2 introduces the Production Cell Case Study. Section 3 presents the COMDES-II model of a distributed Production Cell control system. The verification of system behaviour is discussed in Section 4. Section 5 presents a distributed hardware-in-the-loop simulation facility, which has been used to experimentally validate the software design. A summary of the presented design
and analysis methods is given in the concluding section of the paper.

II. PRODUCTION CELL CASE STUDY

The Production Cell case study is a realistic industrial application, which aims to show the usefulness of formal methods for critical software systems and to prove their applicability to real-world examples.

The problem addressed in the case study belongs to the area of safety-critical systems, as a number of properties must be enforced by the control software in order to avoid injury to people and damage of machines. It is a reactive system, as the control software has to react permanently to changes of the environment. A reduced version of the Production Cell plant has been adopted for this project (see Fig. 2.1).

The simplified Production Cell consists of five machines: a feed belt, an elevating rotary table, a robot with two orthogonal arms, a press, and a deposit belt. All of these machines work jointly to process metal bricks, which are conveyed to a press by the feed belt.

![Deposit belt](image1)

![Robot](image2)

![Arm1](image3)

![Arm2](image4)

![Press](image5)

**Figure 2.1 Top view of the production cell model**

The feed belt transports metal bricks to the elevating rotary table. An electric motor drives the feed belt to move or stop. There is a photoelectric sensor installed at the end of the belt, which is used to indicate if a brick has entered or left the final part of the belt.

The elevating rotary table passes the bricks from the feed belt to the arm1 of the robot. It rotates about 45 degrees and lifts to a level where the arm1 is able to pick up the brick, since the robot arm1 is located at a different level than the feed belt. There are two sensors installed on the table. The first one measures the vertical position of the rotary table, and another one measures how far the table has rotated. Table rotation and motion are effected by two electric motors.

The robot comprises two orthogonal arms and they are set at two different levels. Each arm can retract or extend horizontally so that it can reach the table, press and the deposit belt. Both arms rotate jointly. To grip the bricks, each arm has an electromagnet at the end. The arm1 is responsible for taking bricks from the elevating rotary table to the press, while the arm2 is used for transporting forged bricks from the press to the deposit belt. By default, the arm2 points towards the press and the arm1 is positioned between the table and the press.

There is one sensor on the robot to measure how far the robot has rotated, and an electric motor to rotate the robot. Each arm has a sensor to indicate how long the arm has been extended, a motor to extend and retract the arm and an electromagnet to pick up and drop a brick.

The task of the press is to forge metal bricks. A plate is movable along a vertical axis. Because the robot arms are placed on different horizontal planes, the press plate has three positions. In the lower position, the press is unloaded by arm2, while in the middle position it is loaded by arm1. The brick is processed in the upper position. A sensor is used to measure the vertical position of the press plate. And an electric motor can move the press plate up and down.

The deposit belt transports the bricks unloaded by the robot arm2 out of the production cell. The belt is powered by an electric motor, which can be started up or stopped by the control program. In this simplified version, there is no sensor at the beginning end of the deposit belt to indicate the coming and leaving of the bricks.

Two types of property - safety and liveness properties are considered in this system. The safety requirements are most important: if a safety requirement is violated, this might result in damage of machines, or, even worse, injury of people.

The control program must make sure that various safety requirements are met. Each safety requirement is a consequence of one of the following principles:

- The limitations of machine mobility: the robot, for instance, would destroy itself if rotated too far.
- The avoidance of machine collisions: the robot, for instance, would collide with the press if arm1 would extend too far while pointing towards the press;
- The necessity to keep the metal bricks sufficiently separate: for example: do not put brick on the table, if it is already loaded.

A very strong liveness property for this system is satisfied, if the following requirement is fulfilled:

*Every box introduced into the system via the feed belt will have been forged and will eventually be deposited out by the deposit belt.*

Flexibility is another requirement taken into consideration; namely, the control software has to be open and flexible. The effort for changing the control software and proving its correctness must be as small as possible, when the control system requirements or cell configuration are changed. We believe that a component-based design will satisfy this requirement, as shown in the next section.

III. PRODUCTION CELL CONTROL SYSTEM SPECIFICATION

According to the COMDES–II framework, the control system specification is developed in a top-down fashion. The top level
is defined in terms of actors and their interaction with each other, as well as with their environment. The structural view of the control system provides static information about the interactions between actors and environment. This view is described in an Actor Diagram such as the one shown in Fig. 3.1.

The Production Cell control system consists of five actors: Feed Belt Actor, Table Actor, Robot Actor, Press Actor and Deposit Belt Actor. An Actor is assigned to the physical environment it needs to control. The internal structure of an actor is specified with a function block diagram, e.g. the Table actor shown in Fig. 3.2.

The input signal drivers read the input signals from either plant or communication bus. Messages are received by signal drivers and decomposed into local variables, and sensor signals are read by the physical input drivers. Local signals are further processed by preprocessing function blocks that are either basic or composite function blocks.

The core control part of the Table actor is a hierarchical control unit. It is composed from two controllers: top-level supervisory state machine (SSM) and modal function block (MFB), and second-level SSM and MFB. The top-level state machine has two states, which respond to the manual switch that puts the controller into either On state or Off state. It is executed when triggered by a periodic timing event. However, a transition will take place only when an on/off event is present.

The top-level MFB has two modes for both On (mode 1) and Off (mode 0). The motors of the table are switched off in mode 0. In mode 1, it executes the actual control actions by invoking a sequence of function blocks, as shown in Fig. 3.3.

The second-level state machine (SSM2) is the first function block to execute in mode 1. It performs the main control function of the Table controller. SSM2 determines the current state using the variables provided by the input drivers as well as preprocessing function blocks, and controls the execution of the second-level modal function block (MFB2). The latter executes control actions based on the state indication it receives form the state machine and supplies information to the output drivers, which are then used to generate the output signals of the actor.

The Table actor configuration has been designed as a template, which has been also used with the other actors of the Production Cell control system. We have experienced that this method saves a lot of development time; for instance, the time needed to implement the other four actors has been roughly half of the time spent for the first actor. Furthermore, the possibility of error has been substantially reduced because of the elimination of manual coding. On the other hand, using predefined framework models and components makes it easy to locate implementation errors. It is only necessary to check the system design model, since the implementation follows the principle: “What you design is exactly you implement”.

Figure 3.1 Actor diagram of control system

Figure 3.2 Table Actor

Figure 3.3 Top-level state machine and modal function block of Table Actor
IV. PRODUCTION CELL CONTROL SYSTEM VERIFICATION

The dynamic aspect of the control system is represented by an actor interaction diagram. This is a modified message sequence chart where each lifeline stands for an actor or an environmental unit - Feedbelt, Table, Robot, etc., whose behaviour is specified by the corresponding state machine (a fragment of the diagram is shown in Fig. 4.1). Furthermore, an actor state machine is analyzed in conjunction with the corresponding environment state machine. These constitute a pair, i.e. a subsystem that is modelled as a parallel composition of the two state machines, which interact by exchanging control and feedback signals.

![Diagram of a pair consisting of Table Environment and Table Actor](image_url)

**Figure 4.1** A pair consisting of Table Environment and Table Actor

A complex control system may consist of a number of independent and/or interacting subsystems, as is the case with the Production Cell Case Study (see e.g. Fig 4.1). Therefore, it seems natural to use a pairwise compositional approach to the problem of sequential control system verification as advocated in [7], whereby local properties are specified with respect to subsystems, i.e. plant-controller pairs, as well as pairs of interacting subsystems, and global properties are represented as a conjunction of local properties.

It can be argued that a safety property is entirely related to the operation of a particular subsystem, or possibly - a number of subsystems, whereby the global property is defined as a conjunction of local properties. For example, the property "A brick must never fall down during plant operation" may be formulated in the context of individual subsystems such as Feedbelt, Table, Robot, etc., as well as the Production Cell as a whole, and the global property can be specified as a conjunction of the corresponding local properties.

Likewise, a global liveness property may be investigated along a specific trajectory of subsystem reactions and interactions, which is determined by the operational specification of the system, i.e. the actor interaction diagram. The latter specifies a distributed transaction involving a number of interacting subsystems, e.g. the sequence of actions and interactions triggered by a transaction start-up event, such as the arrival of a brick on the feed belt. The start-up event triggers a reaction in a subsystem resulting in the generation of an output signal that triggers a reaction in a second subsystem; the generated output signal triggers a reaction in a third subsystem, and so on - as specified by the interaction diagram, until the final control signal is generated.

The proposed idea is that the analysis should check the behaviour of each subsystem in order to prove that the arrival of a specific input signal (event) is followed by the expected output signal, which is then applied as an input signal to the next subsystem, whose behaviour is checked in the same manner, and so on. Ultimately, the system reaction will be correct and the specified control signal – generated, if each of the subsystems involved behaves correctly. This means that a system-wide liveness property, as formulated above, can be represented once again as a conjunction of local properties, i.e. liveness properties of the subsystems taking part in the transaction.

The main advantage of the proposed analysis method is that it reduces considerably the problem of state explosion, by limiting the explored state space to that of the product state machine of the pair under investigation. At the same time, this method takes into account the tight interaction between plant and controller state machines, which is typical of sequential control systems.

With this method, model checking may be carried out using existing techniques and tools, e.g. Uppaal [5], whereby a property is specified by a temporal logic formula, which is then checked in the state space of the product state machine. To that end, the actor interaction diagram must be translated into Uppaal automata. A number of rules have been defined to perform this translation:

1. Environment: each environment is modeled as a process in Uppaal
2. Actor: each actor is modeled as a process in Uppaal
3. Sensor Signal: this is a trigger sent from the environment to the control actor. It is modeled as variable in Uppaal:
   ```
   int [0,1] env_feedbelt_sensor;
   ```
   That variable is included only in a guard on some transition of the control automaton. The environment process will set this variable and the controller will read it. No synchronization is necessary here, as the controller does not have to be blocked to just wait for a specific value of the sensor.
4. Control Signal: This is a response from actor to environment. It is modeled as broadcast channel:
   ```
   urgent broadcast chan env_feedbelt_motor_on;
   ```
   The controller generates the control signal and the environment waits in a state to receive that signal. After the control signal is received, the state of the environment may be changed. In Uppaal a transition with an emit-synchronization on a broadcast channel can always fire if the guard is satisfied, no matter if any receiving transition is enabled. But a receiving transition, which is enabled, will synchronize. So, the broadcast channel can be used to model the control signal passed form actor to plant. The use of urgent channels makes it possible not to delay the sender in the source state if it is possible to trigger synchronization over an urgent channel.
5. Global Signal: this type of signal is used for actor-to-actor interaction, such as handshaking. It is modeled as a channel:
   ```
   urgent chan ready_for_feed;
   ```
Both sender and receiver are blocked if they are not at the rendezvous points.

6. The step between sensor signal and control signal or global signal is modeled as a state in the automaton.

The Uppaal system model consists of one or more concurrent processes modeling the actors and environment units. In the real life, the controller is much faster than the environment, because it has to react as soon as it gets input from the environment. To model this feature in Uppaal, the controllers must have higher priority than the environment:

```
system Env_FeedBelt, Env_Table
  < FeedBeltController, TableController;
```

Uppaal employs interleaved process execution. Therefore, if the environment process and the controller process have the same priority, it is possible to have the following scenario: The controller is blocked waiting for the sensor signal to become true in some state, and after some time the environment sets it to true. Since the environment has the same priority with the controller, after setting the sensor, the environment process (or other processes) may continue running, and the controller will be still blocked. Then the controller could miss this sensor signal if another process takes long time.

However, if priority is used and the controller process has higher priority, the environment process can run only if the controller process is blocked. In the above scenario, the controller process will be blocked while waiting for the sensor signal to become true, and the environment process can run until it sets the sensor signal to true. After the sensor signal is set, the controller process will not be blocked, so it will run and block the environment process. And since the communication is urgent, i.e. it has to fire immediately whenever possible, the controller will never miss the sensor signal.

By applying these rules, the actor lifeline in the actor interaction diagram can be translated into an Uppaal process model (e.g. Fig. 4.2). The environment units such as Feedbelt, Table, Brick and so on, need to be modeled as process respectively according to the abstraction of the real plant.

![Figure 4.2 Table Actor model in Uppaal](Image)

In the real world, we could judge if the controllers are designed correctly by observing the behaviour of the controlled objects. In this case, correctness is implied by the proper behaviour of bricks and environments; thus properties are specified using elements of brick and environment processes. For instance, we could check the property: Is it possible that a brick is moving with table and another is moving with feed belt?

The safety requirements are summarized as following:
- R1: Keep bricks sufficiently distant
- R2: Avoidance of machine collisions
- R3: Limitations of machine mobility

For R1, the bricks can not overtake each other, i.e. it is necessary to show that their locations in the system are occupied under mutual exclusion. The property can be formulated as a statement that only one brick at a time can be on the table, assuming that there are five bricks in the system:

```
A[] (brick1.OnTable imply not (brick2.OnTable or brick3.OnTable or brick4.OnTable or brick5.OnTable))
```

For R2, we could check a statement specifying that arm1 can not wait in the press area when the press is moving up or down close to the process area:

```
A[] not (Env_Robot.Stop and Env_Robot.position==1 and Env_Press.DMovingInProcessArea),
```

where the local variable “position” represents robot position.

For R3, the table never enters the damage state:

```
A[] not Env_Table.Damage
```

It is not necessary to check this property against the entire system which has a big state space, as the property is only locally related to the table actor and the table environment. It can be checked within the Table pair leading to a smaller state space of the verification model.

However, the Table actor needs to synchronize with the Feed belt actor and the Robot actor. In this case, the complete models of these two actors are not necessary as the Table actor can synchronize with some fake actors that provide only the required handshake signals. Therefore, for this local property the system consists of only the Table environment, Table actor, fake Feed belt and Robot actors. Even the brick process is not necessary.

Likewise, similar properties for the robot and press can also be checked locally by providing fake actors.

```
A[] not Env_Robot.Damage
A[] not Env_Press.Damage
```

Consequently, a global property like “The system never enters a damage state” does not have to be checked on the entire system at all. The global property is actually a composition of the local properties. If all the local properties are verified to be true, this global property will be true as well:

```
A[] not ( Env_Table.Damage or Env_Robot.Damage or Env_Press.Damage ) =
A[] not Env_Table.Damage AND
A[] not Env_Robot.Damage AND
A[] not Env_Press.Damage
```
V. Hardware-in-the-Loop Simulation and Related Experiments

Hardware-in-the-loop simulation can be used to develop and test embedded control systems. It is particularly efficient with complex systems, whenever it is very costly or impossible to use the plant itself in the process of software development [4].

The Production Cell has been simulated on a PC, which performs the same function as the real plant: it responds to the motor controlling signals and sends out sensor signals to the embedded control system, using a dedicated interface implemented with National Semiconductor process I/O boards. The control system is a distributed application running under the HARTEX kernel [2], built on top of a Controller Area Network coupled to a computer model of the Production Cell.

The OpenGL package has been chosen to build a graphic plant simulator. This is a software interface for graphics applications, which provides a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the frame buffer.

Building the Production cell simulation is very similar to building a house with basic elements like brick, window, door, etc., as all the complex models consist of some basic elements such as box, cylinder, octagon, etc.

To make the plant model “alive”, it is necessary to couple graphical objects to the corresponding models of plant dynamics. The plant has to react appropriately when receiving certain control signals, and set sensors when the graphical model is in some situation. To that end, state machines are associated with graphical models, so as to receive control signals and react graphically by moving some objects on the screen and set sensors via the plant/control system interface.

The verification of the actor interaction diagram has helped us discover a fault in the initial system design. To see if the design was efficient, we checked whether it was possible for a brick to move with arm1 of the robot while another one was moving with arm2 at the same time. The property was expected to be true, but it turned out to be false. By analyzing the counter-example given by Uppaal, we found that the system did not support the above feature in the present version of the actor interaction diagram. This defect was fixed by designing another version of the robot actor.

Subsequent hardware-in-the-loop experiments have validated the developed Production Cell control system and have demonstrated the feasibility of our software design method.

VI. Conclusion

The paper has presented the COMDES-II framework in the context of the Production Cell case study. The framework supports both open system architecture and predictable behaviour. It defines a consistent set of executable models specifying relevant aspects of system structure and behaviour within the domain of distributed embedded systems for mechatronic applications.

System structure is specified statically in terms of distributed embedded actors that communicate with each other by exchanging labeled messages (signals). Signal-based communication provides for transparent interaction between actors, independent of their allocation onto network nodes.

Actors are built from reconfigurable components: signal drivers as well as basic, composite, modal and state machine function blocks. The developed components have been experimentally validated through hardware-in-the-loop simulation of the developed Production Cell control system.

The dynamic view of the system is represented by an actor interaction diagram, which has been translated into an Uppaal verification model. The model consists of paired state machines that specify explicitly the actor and its corresponding controlled environment. Using the plant-controller pairs, the Uppaal model can be checked in full size or partially - within a sequence of chained reactions, concluding the correctness of system properties from local properties of the plant-controller pairs involved in the transaction.

REFERENCES


Figure 5.1 Hardware-in-the-loop simulation experiment