Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

A Control Software Development Method Using IEC 61499 Function Blocks, Simulation and Formal Verification

Goran Čengić **Knut Åkesson**

Department of Signals and Systems
Chalmers University of Technology, Sweden
{*cengic, knut*}*@chalmers.se*

Abstract: A new control software development method is presented. It uses IEC 61499 function blocks for control software programming and provides tools for simulation, execution, automatic model generation and formal verification of the control code during the development. Simulation and execution are supported by the same tool, the Fuber runtime environment. Formal modeling is done using extended finite automata (EFA) and an automatic model generation tool. Formal verification shows the behavior of the closed-loop system. i.e. when control code is executed against the model of the process. The model can use a non-deterministic execution control chart (ECC) in the process model block. The control code and the process model are expressed using the IEC 61499 language in order to avoid maintenance of the process model and control code in different languages, thus making it easier to use the formal verification in the control software development.

Keywords: Control Software Development, IEC 61499, Extended Finite Automata, Fuber, Supremica.

1. INTRODUCTION

Manufacturing and processing systems are typically controlled by a distributed control system. Developing distributed manufacturing and process control systems is time-consuming and error-prone because the environment is typically heterogeneous, i.e. consisting of hardware from multiple vendors. This often implies that different languages and development tools are used to develop the software for control functions. In 2005 the International Electrotechnical Commission (IEC) approved a new standard called IEC 61499 (IEC [2005]) to facilitate the development of distributed control systems in heterogeneous environments.

Industrial control systems have generally high requirements on reliability and tight timing constraints for the control functions. Thus the control software is typically written in special purpose languages defined in the IEC 61131 standard (IEC [1993]) and executed on special purpose hardware called Programmable Logic Controllers (PLCs). The IEC 61499 provides the next generation special purpose language that extends the existing IEC 61131 standard.

A number of development environments for IEC 61499 have emerged. These include CORFU (Thramboulidis and Tranoris [2004]), OOONEIDA-FBench and ISaGRAF. There are also a number of IEC 61499 runtime environments in existence, some of those are Fuber, RTSJ-AXE (Thramboulidis and Zoupas [2005]) and RTAI-AXE (Doukas and Thramboulidis [2005]). The last two runtime environments are focusing on the real-time execution of the IEC 61499 applications.

Current research on the IEC 61499 standard has focused on architectures and methods for development of the control applications (Thramboulidis [2002], Vyatkin et al. [2005], Čengić et al. [2006a]), performance analysis of runtime environments (Ferrarini and Veber [2004]), usability and interoperability of the implementations (Sünder et al. [2006]), execution semantics (Vyatkin et al. [2007]) and formal verification of the applications.

Substantial work has been done on the formal verification of the IEC 61499 applications. In Vyatkin [2006] applications are modeled using net condition/event systems. Dubinin et al. [2006] presents a method for application verification based on Prolog language. In Hagge and Wagner [2005] a new modeling language based on Petri Nets is introduced. In Dubinin and Vyatkin [2006] a formal semantic model of IEC 61499 function blocks is presented. In Stanica and Guéguen [2004] the applications are modeled using the timed automata. No details of the runtime environment are modeled. Without the runtime model the verification does not represent the execution model implemented by the runtime and therefore does not give enough information about block scheduling order since that order is not specified in the standard. In Čengić et al. [2006b] the applications are modeled using finite automata that also capture many details of the runtime environment behavior, including block scheduling order.

Most of the reviewed methods for formal verification focus on the modeling and verification of the control code by it self. But to be more useful the formal verification should

provide the information about the behavior of the closed-loop system when the code is executed together with the process. To obtain that information a model of the process under control is necessary.

Usually the control code reacts on the events that occur in the process under control. To react correctly some assumptions have to be made about the sequence of the events that occur in the process, i.e. the control code assumes a certain behavior from the process. But in the reality the process sometimes departs from the anticipated behavior, e.g. when something goes wrong. To ensure the correct behavior of the closed-loop system the model of the process under control has to be able to express such unanticipated behavior of the process. One suitable way to do that is to introduce non-deterministic behavior within the process model. The non-deterministic behavior then represents that after a sequence of events the actual state of the process is not known and the formal verification can show how the control code will react to that.

Introducing the non-deterministic behavior within the process model is possible by using a suitable modeling language. The formal verification may then take the formal model of the control code and the non-deterministic model of the process and provide the behavior of the closed-loop system. Since the control code is written using the IEC 61499 language it would be useful to also allow the non-deterministic process model to be expressed using the IEC 61499 language. Then the formal models for the verification of the closed loop behavior may be automatically generated. This way the necessity to maintain the model of the process in a separate language from the control code language is avoided, making it easier for the control code developer to use formal verification.

This paper presents a development method for the control software that supports development of the control code together with the development of a (possibly non-deterministic) process model, all expressed using the IEC 61499 language. The method provides a software tool for automatic generation of formal models from the IEC 61499 application that represents the closed-loop between the control code and the process model. The formal verification of the closed-loop system is done using the discrete event systems tool Supremica (Åkesson et al. [2006]). The method supports the simulation of the closed-loop system in addition to the formal verification.

The paper is organized as follows. In Section 2 the basics of IEC 61499 standard are introduced. In Section 3, the proposed development method is presented and all the development steps are described. An example application is used to demonstrate the use of the proposed method in Section 4. Finally some conclusions are presented in Section 5.

2. IEC 61499

This section briefly introduces some of the terminology used in the paper. The software architecture defined by the IEC 61499 standard (IEC [2005]) is based on functional software units called function blocks. The basic function block type is the main entity. In Fig. 1(a) the anatomy of a basic function block type is shown. The left side of

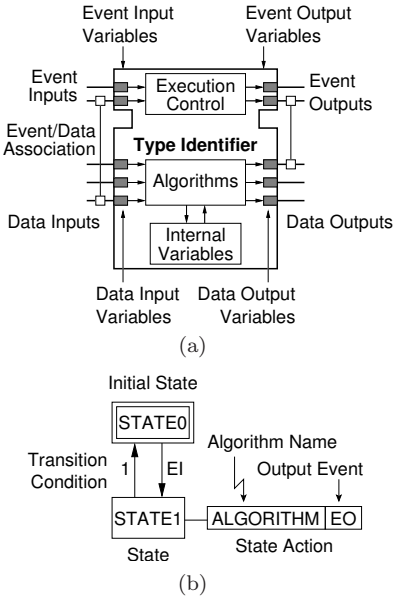

Fig. 1. **(a)** Anatomy of a basic function block. **(b)** An example of an execution control chart (ECC).

the block contains the event and data inputs while the right side contains the event and data outputs. The basic function block executes algorithms based on the arriving events and generates new events that are passed on when the algorithms finish execution. The algorithms use data associated with the incoming events to update internal variables and produce output data. When an algorithm has terminated an output event is generated triggering another function block for execution.

The Execution Control Chart (ECC) of a basic function block determines which algorithm to execute based on the current input event and values of input, output and internal data variables, see Fig. 1(b). When a state is entered each action associated with the state is executed once and the ECC stays in the state until a condition for entering another state is fulfilled. The conditions upon which transitions occur are Boolean expressions involving input events and input, output and local data variables. A special case of a transition condition is the transition labeled with "1", which means that it is always true and is taken as soon as all the actions of a state are executed.

The example ECC in Fig. 1(b) states that if it is in `STATE0` and input event `EI` is received, the ECC transfers to state `STATE1` and schedules the algorithm named `ALGORITHM` for execution. After `ALGORITHM` has terminated the output event `EO` is generated, and the ECC returns immediately to state `STATE0` since the transition condition is "1" (true). Basic function blocks are connected together by event and data connections into function block applications. The applications can be executed using a runtime environment that implements the execution model defined by the standard.

3. THE DEVELOPMENT METHOD

The development method that is proposed is shown in Fig. 2. It consist of several activities and support tools that are discussed in this section. The method is based on

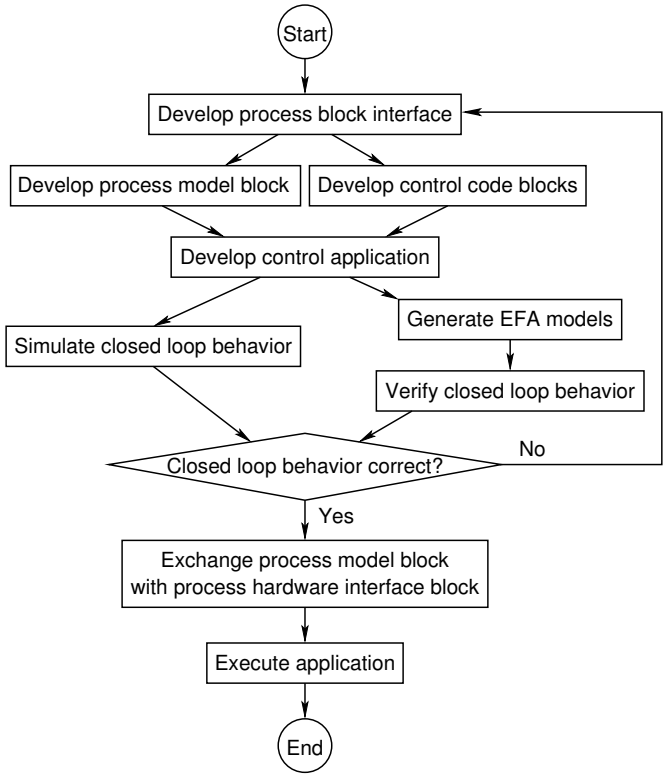

Fig. 2. Activity flow diagram for the development method.

the separation of the process model and the control code, therefore the first step in the development of the control application is the design of the interface of the function block that will be used for communication with the process under control. Any function block that implements the specified interface is called a plant block during the development process.

When the interface specification is finished the basic function block that implements a model of the process can be developed. Using the ECC and the algorithms of a basic function block the developer can model the behavior of the process at any level of abstraction needed for the application. Although a number of model blocks may be useful for the application only one of the blocks should be used for each iteration of the development process. At the same time as the model blocks are developed the control code blocks can be developed since the process interface specification is finished. The control code blocks implement the control logic for the application.

When the control code blocks and one of the model plant blocks are finished they can be put together in an application that is used for verification and/or simulation. The simulation activity is supported by the Fuber runtime environment and requires only that the application is finished. The verification activity is supported by the Supremica tool and requires that the formal models are generated from the application prior to the use of the verification tool. The formal models are automatically generated using a different software tool.

The simulation and verification steps of the development process show if the desired closed loop behavior is correct or not. If it is not correct the development process calls for a new iteration of development of the process interface and the application blocks. If the closed loop behavior is correct the development may proceed with exchange of the model plant block with a plant block that communicates with the hardware process and the application can be executed. In the following subsections some of the method's steps are discussed in more detail.

3.1 Simulation

The simulation and execution of the function block applications is supported by a free software IEC 61499 runtime environment called Fuber. Fuber is able to open IEC 61499 compliant applications and execute them. The applications have to be supplied in a XML-file that is compatible with the standard.

The Fuber acts like simulator when the application executes using the plant block that implements a model of the process. This allows the closed loop behavior to be tested using the same runtime environment that may execute the finished control code.

In the case when the model plant block is non-deterministic the simulation result will only show what happens for a single scenario since the Fuber will always make the same choice from the available choices for a given event depending on how the model plant block type has been declared. To simulate how the application will react for each available choice a new model plant block has to be developed. The new deterministic model can be implemented in such a way that each available choice is made following some pattern. Now the simulation shows how the application will react to each choice but it does not capture how the application will react to the different sequences of the choices.

For some applications it is not feasible to test all the different sequences of the choices for a given event using the simulation. In those cases the formal verification may be the only way to ensure the correct application behavior. On the other hand for the large examples the formal verification using non-deterministic process model may take too much time, or the models may become to big, then the only way to get some assurance of the correct application behavior is to identify most common scenarios and simulate the application behavior for those.

3.2 Model Generation

The modeling of the application execution assumes that the application is executed in a runtime environment that implements the execution model based on the sequential hypothesis as presented in Vyatkin et al. [2007]. The Fuber runtime environment is compatible with that execution model based on the presentation of the Fuber modeling in Čengić et al. [2006b].

The modeling approach used in the automatic model generation tool for the new development method is based on the modeling presented in Čengić et al. [2006b] and extended to support more details of the execution. The formal verification results may therefore be valid for all IEC 61499 runtime environments that are compatible with the execution based on the sequential hypothesis, including the Fuber.

A software tool is used to automatically translate function block application into a set of extended finite automata (EFA) (Sköldstam et al. [2007]) that can be analyzed using the Supremica tool. The model generation tool produces a model that captures both the behavior of the application and the behavior of the runtime environment while it executes the application.

The non-deterministic behavior in the model plant blocks is supported by allowing the ECC of those blocks to be non-deterministic. The non-deterministic ECC is an ECC that has transitions with the same Boolean expression from a single source state but to different destination states. The non-deterministic ECCs are translated into non-deterministic EFA models by the model generation tool.

Currently the model generation tool supports only integer variables in the application, and since the generated models are finite the model generation tool imposes an upper and lower bound on the integer variables. These bounds are user configurable. Another limitation is that basic block algorithms may only contain assignment statements, i.e. variable = expression. All applications that can be loaded and executed using the Fuber runtime, and also conform to the limitations above can fully automatically be converted into EFA models.

3.3 Formal Verification

The formal verification is supported by the Supremica tool. Supremica is a general purpose tool for synthesis and verification of discrete event systems. It has a graphical user interface for the input of models and a range of algorithms implementing supervisory control theory methods for synthesis of supervisors and verification of the models.

4. EXAMPLE

In this section an example control application is developed to illustrate the use and the benefits of the proposed method. The process under control is shown in Fig 3. The goal is to sort steel balls that are placed in a queue at the process gate. Each ball is let into the process by the gate and then it moves to the first lift. The lift brings the ball to the measuring station. Once the size of the ball is measured it is pushed out and it moves to the second lift. Depending on the size, the ball is transported to the first or second level. Once at the destination level the ball is pushed out of the lift and it arrives at the position where it will be picked up by the rotating hand for transport back to home position in front of the gate. Two types of balls are handled by the process, the small and the big ones. The small balls should be moved to the first level and the big balls to the second level.

To control the process an IEC 61499 function block application has been developed using the proposed method, see Fig. 4. For the example let us assume that this application is the result of the "Develop control application" method activity of the first iteration in the development process. The first versions of the plant block interface, model plant block implementing the interface, and the control code blocks have been developed and connected together in the application.

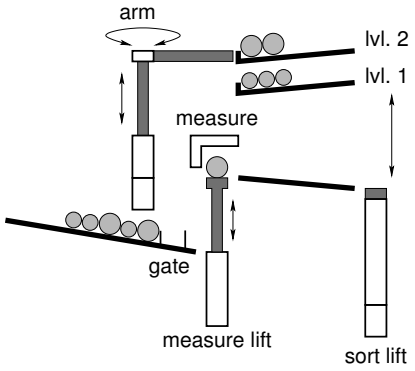

Fig. 3. The ball sorting process used in the example.

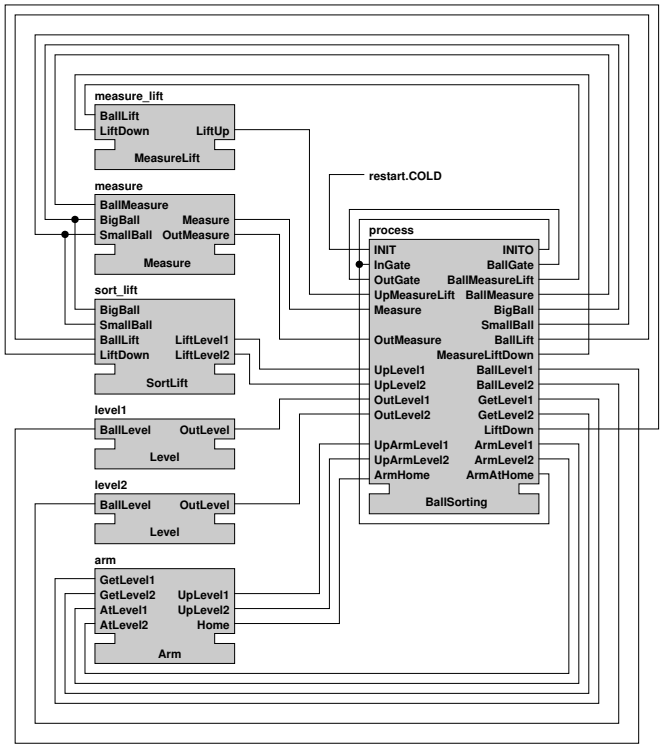

Fig. 4. Function block application for the ball sorting process.

The plant block interface can be seen in the Fig. 4 as the interface implemented by the "process" instance of the "BallSorting" basic function block type. The event inputs of the block represent all the actuators that can be set in the process. Since only events are used, the implementation of the input event is that it sets the actuator high and resets it to low once the actuators task is done. The event outputs represent the process sensors' rising edges.

The model of the process implemented by the basic function block type "BallSorting" is very simple. Each station in the process is modeled separately, so when the event input "InGate" is received the "BallGate" event output is sent indicating that the ball has entered the gate. When the "OutGate" event input is received the "BallMeasure-Lift" is sent out to indicate that the ball has arrived in the lift that will take it to the measuring station. In general, a sensor that is activated by the ball's free movement after

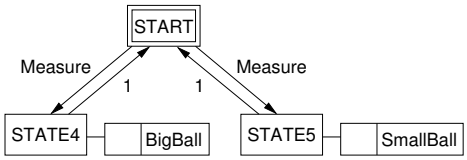

Fig. 5. The part of the model block's non-deterministic ECC showing the transitions when receiving the "Measure" event.

an actuator has been set, will generate event output as soon as the actuator is set.

It is the goal of the control application to transport the ball through the process and sort it according to its size. The basic blocks implementing the control application are shown to the left of the "process" block. These blocks are connected in a closed loop with the "process" block emphasizing the separation of the control code from the plant. To keep the example simple only one ball may be in the process.

The application starts when the "restart.COLD" event is generated by the Fuber runtime environment and it initializes the model block. The model sends out the "INITO" event that is connected to the "InGate" input event. The "InGate" event lets one ball into the gate and sends out the "BallGate" event to the "OutGate" input that in turn closes the gate inlet and opens the gate outlet so that the ball may enter the process. Once the ball has rolled into the first lift the "BallMeasureLift" is sent out to the "measure_lift" block of the "MeasureLift" type that instructs the lift to move up the ball to the measuring station. The ball continues like this all the way through the process until it is at the home position.

The correct behavior of the application may now be tested using formal verification or simulation. The simulation is done by simply running the application in Fuber and evaluating its behavior. The result of the simulation shows the application behavior for a single sequence of the balls at the gate. The behavior of the application for any sequence of the balls at the gate can be obtained by analysing the result of the formal verification.

For formal verification the EFA models of the application are generated using the model generation tool prior to verification. Since the model generation tool supports non-deterministic function block ECCs the "process" block is implemented so that the event input "Measure" takes the ECC to two different states, see Fig. 5. In one state the "SmallBall" event output is sent while in the other the "BigBall" event output is sent.

The non-deterministic ECC of the process model captures the possibility of both the small and the big ball being measured. Since it is not known which type of the ball will enter the system at any time this allows the verification to check both possibilities at the same time using a set of EFA models that are generated only once.

To verify the closed loop behavior of the application some EFA models of the correct behavior need to be entered into the Supremica. Each model can specify a different aspect of the correct behavior. In Fig. 6 one such EFA model is shown. This model checks that the control application

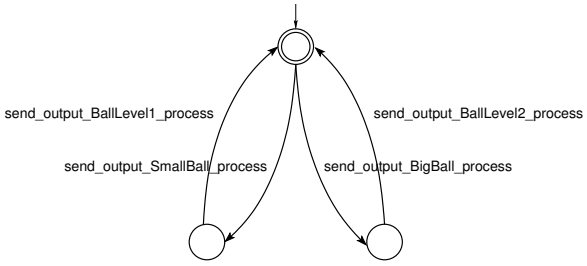

Fig. 6. The specification of the correct behavior for the formal verification of the application.

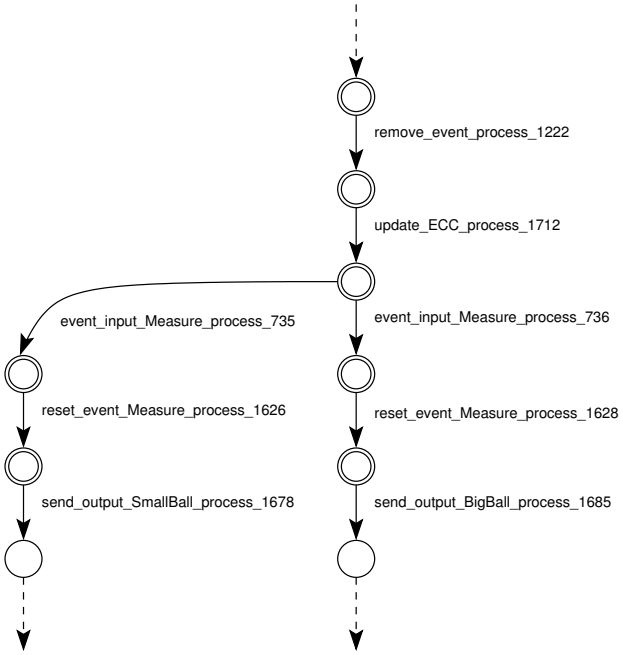

Fig. 7. The part of the complete closed loop behavior of the application. It shows what happens when the ball is measured, the left branch for the small ball and the right branch for the big ball.

lifts each small ball to the first level and each big ball to the second level. The only marked state specifies that the application must always be able to reach back to the initial state which means that each ball is sorted correctly.

The verification that is done is called the non-blocking verification. This verification type checks that the closed-loop behavior of the system can reach a marked state from any other state, e.g. the application finishes correctly. If the system passes this form of verification it will not get stuck in a single unmarked state during execution, so called dead-lock, nor will it get stuck into a loop over a subset of unmarked states, so called live-lock.

The complete closed-loop behavior of the application can be calculated. A part of the result of such a calculation is shown in Fig. 7. The part shown represents what happens in the system when the ball is measured. Since the non-deterministic ECC of the process model block allows for both the big ball and small ball to be reported that is shown in the resulting behavior of the system. The left branch shows what happens if the small ball is measured while the right branch shows what happens if the big ball is measured. Each input to the process is thus covered in the

formal verification and so is the control code that reacts to that process input.

In Fig. 7 it is shown that the states become unmarked after the process sends out the event that reports the size of the ball. This is the consequence of the specification EFA model since all the states in the automatically generated application model are marked. The states will remain unmarked in each branch of the closed-loop model until the process sends out events reporting that the ball has arrived to the specified level.

When the correct behavior of the application has been achieved the plant model block can be exchanged with a block that communicates with the hardware of the process and executed.

5. CONCLUSION

The proposed development method provides the means for closed-loop control system development by integrating process modeling and control code development that use the same language. The method makes development of control code possible without ever running it against the real hardware until it is ready by supporting the simulation and verification of the closed-loop system. This leads to shorter development time and less economic loss in cases when the control application is behaving incorrectly and destroying the controlled equipment while it is being tested against the real process.

REFERENCES

Goran Čengić, Oscar Ljungkrantz, and Knut Åkesson. A framework for component based distributed control software development using IEC 61499. In *Proceedings of the 11th IEEE International Conference on Emerging Technology and Factory Automation*, pages 782–9. IEEE, September 2006a.

Goran Čengić, Oscar Ljungkrantz, and Knut Åkesson. Formal modeling of function block applications running in IEC 61499 execution runtime. In *Proceedings of the 11th IEEE International Conference on Emerging Technology and Factory Automation*, pages 1269–76. IEEE, September 2006b.

George Doukas and Kleanthis Thramboulidis. A real-time linux execution environment for function-block based distributed control applications. In *Proceedings of 3rd IEEE International Conference on Industrial Informatics*, pages 56–61. IEEE, August 2005.

Victor Dubinin and Valeriy Vyatkin. Towards a formal semantic model of IEC 61499 function blocks. In *Proceedings of the 4th IEEE International Conference on Industrial Informatics*, pages 6–11. IEEE, August 2006.

Victor Dubinin, Valeriy Vyatkin, and Hans-Michael Hanisch. Modelling and verification of IEC 61499 applications using Prolog. In *Proceedings of the 11th IEEE International Conference on Emerging Technology and Factory Automation*, pages 774–81. IEEE, September 2006.

Luca Ferrarini and Carlo Veber. Implementation aproaches for the execution of IEC 61499 applications. In *Proceedings of 2nd IEEE International Conference on Industrial Informatics*, pages 612–7. IEEE, June 2004.

Nils Hagge and Bernardo Wagner. A new function block modeling language based on petri nets for automatic code generation. *IEEE Transactions on Industrial Informatics*, 1(4):226–37, November 2005.

IEC. IEC 61131 programmable controllers—part 3: Programming languages. Technical report, International Electrotechnical Commission, 1993.

IEC. IEC 61499-1: Function blocks—part 1: Architecture. Technical report, International Electrotechnical Commission, 2005.

Markus Sköldstam, Knut Åkesson, and Martin Fabian. Modelling of discrete event systems using finite automata with variables. In *Proceedings of the 46th IEEE Conference on Decision and Control*, pages 3387–92. IEEE, Dec 2007.

Marius-Petru Stanica and Hervé Guéguen. A timed automata model of IEC 61499 basic function blocks semantic. In *Proceedings of the 7th International Workshop on Discrete Event Systems*, pages 385–90. IFAC, September 2004.

Christoph Sünder, Alois Zoitl, James H. Christensen, Valeriy Vyatkin, Robert Brennan, Antonio Valentini, Luca Ferrarini, Thomas Strasser, Jose Lastra, and Franz Auinger. Usability and interoperability of IEC 61499 based distributed automation systems. In *Proceedings of the 4th IEEE International Conference on Industrial Informatics*, pages 31–7. IEEE, August 2006.

Kleanthis Thramboulidis. Development of distributed industrial control applications: The CORFU framework. In *4th IEEE International Workshop on Factory Communication Systems*, pages 39–46. IEEE, 2002.

Kleanthis Thramboulidis and Chris Tranoris. Developing a CASE tool for distributed control applications. *Journal of Advanced Manufacturing Technology*, 24(1–2):24–31, July 2004.

Kleanthis Thramboulidis and Alkiviadis Zoupas. Real-time Java in control and automation: A model driven development approach. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 39–46. IEEE, September 2005.

Valeriy Vyatkin. Execution semantic of function blocks based on the model of net condition/event systems. In *Proceedings of the 4th IEEE International Conference on Industrial Informatics*, pages 874–9. IEEE, August 2006.

Valeriy Vyatkin, James Christensen, and Jose Lastra. OOONEIDA: An open, object-oriented knowledge economy for intelligent industrial automation. *IEEE Transactions on Industrial Informatics*, 1(1):4–17, February 2005.

Valeriy Vyatkin, Victor Dubinin, Carlo Veber, and Luca Ferrarini. Alternatives for execution semantics of IEC61499. In *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, pages 1151–6. IEEE, June 2007.

Knut Åkesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 384–5. IEEE, July 2006.