

# Applying the handler-based execution model to IEC 61499 basic and composite function blocks

Nils Hagge and Bernardo Wagner, *Member, IEEE*  
Institute for Systems Engineering, Real-Time Systems Group  
Universität Hannover  
Appelstraße 9A, D-30167 Hannover, Germany  
{hagge|wagner}@rts.uni-hannover.de

**Abstract**—We defined the so-called “Handler-based execution model” as a real-time capable execution model for CNet [1] [2]. Traditional controller implementations typically consist of big loops with a fix sequential execution scan order. This was not appropriate for Petri-net based CNet that is characterized by a high degree of concurrency and locality. Our execution model is purely based on events to dynamically schedule the evaluation of firing conditions and avoids unnecessary calculations. There are no global event queues. Event-flow is handled locally which is beneficial for multithreaded and/or distributed platforms. This execution model is part of a full automatic code generator that translates CNet controller models into (Real-time) Java code. Examining IEC 61499 revealed that function blocks networks similarly feature concurrency and locality, but that execution environments for function blocks are still developed based on traditional fixed sequential scan approaches.

This paper will introduce the main concepts of the “Handler-based execution model” and show its possible application to IEC 61499 function blocks for automatic code generation.

**Index terms**—IEC function blocks, Petri nets, execution model, automatic code generation

## I. INTRODUCTION

SEVERAL modeling languages exist with different emphasis to design controller applications. But there is still a gap in the corresponding tool-chains when it comes to the implementation level. On the design level description languages are intended to be intuitive and shall abstract from implementation details. Analytic methods are provided to identify design errors at an early stage during the design phase. However, the actual implementation of the verified or at least tested model is often done manually. Since there is in general no formal way to prove that the hand-coded software exactly realizes the intended behavior, several software engineering techniques have been developed such as pair programming or code reviews in order to improve software quality. But all these techniques are still error-prone since they fully depend on personal experience of the implementers and do not incorporate formal methods. A complete modeling framework for developing safety critical control applications needs to assure that the resulting control code complies to the verified model. This can only be

achieved by formalizing and automating the creation process of the application code and restricting the designer to work with the system model only.

CNet was introduced by Wurmus [1] as a component-based conception for designing modular, concurrent and distributed control systems. The interfaces of CNet components are formed by special language elements derived from the Petri net theory. The components can be reused and their event-discrete behavior is described by a special class of colored Petri nets with arc timing, called PNet. Details can also be found in [2] [4] [5].

In order to automatically generate real-time capable controller application code, we sought for an execution model that precisely implements the behavior of CNet and takes advantage of the characteristics of the Petri net based behavioral descriptions. Petri net models are expressive in terms of concurrency and locality, which should be respected by the execution model to be defined.

Jörns *et al.* [8] proposed a method to automatically derive programs for programmable logic controllers (PLC) from Petri nets. The architecture of such PLC systems enforces a certain execution model that is referred to as “scan-based” execution model [7]. Following the scan-based execution model, the control programs consist of a predefined sequence of instructions. These instructions are processed always the same way from top to bottom. Fig. 1 shows the typical PLC cycle [6] [3] that consists of reading all inputs to the controller into data space, processing the instructions

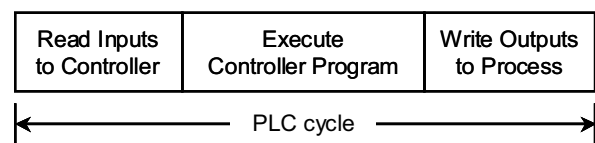


Fig. 1. Execution model of standard programmable logic controllers

that work on these data, and finally copying the calculated result values to the outputs to the process. This PLC cycle is infinitely repeated from the beginning, no matter whether the inputs change or not. Since the evaluation order of the firing conditions is crucial, Jörns *et al.* use simulation to determine the right sequential order. This technique was improved by Frey [3], who presented an analytical approach. The main drawback of the generated PLC programs due to

the restricted PLC architecture is that most of the instructions are executed unnecessarily in each PLC cycle, because the calculations lead to the same results as in the preceding cycle, if the inputs have not changed. We observed that, even if one input changes, only a few number of transitions fire and a small number of outputs will produce different values. This discrepancy between necessarily and unnecessarily evaluated firing conditions is caused by the disregard of the *locality* of transitions and their pre- and post-places expressed by the connecting arcs. Instead of reading only inputs whose values have changed and reprocessing only instructions that depend on them, the PLC takes at the beginning of each cycle (Fig. 1) a *global* snapshot of the process values and reruns all instructions.

Another important point concerns concurrency. Although Petri nets explicitly allow the modeling of concurrent activities, this is not exploited and pure sequential code is generated instead.

For CNet we defined an event-discrete execution model, that is suitable for a fully automated generation of real-time capable code and takes into account the main characteristics of CNet system models: concurrency and locality. This execution model, called *handler-based execution model*, features exploitation of modeled concurrency and assigns events to all changes of system-state describing values, which allows for reducing the need for synchronization of concurrent activities to the direct access to shared resources. This way of decoupling concurrent execution threads provides the basis for partitioning the control application code, which again marks the key for a distribution of the control application to several process computers or controllers.

We argue that the characteristics of IEC 61499 function block networks are also stamped by concurrency and locality as well as by an event-driven data propagation. However, many effort is spend to implement function block (FB) based applications in a PLC-like scan-based fashion. In [7] an application generator is introduced that transforms FB networks into a fix sequence of conditionally executed subroutines. This approach addresses the problem of unnecessarily processed function blocks by checking its event-inputs and skipping to the next one if applicable. However, the execution order is still fix and the authors acknowledge “Even though the Pointe Controller is considered to have advanced functionalities compared to a typical PLC, its program execution follows the PLC tradition, thus also implementing a scan-based execution model” [6]. The execution order is arranged according to the event-flow of the FB network, but this is not always possible, since the event-flow is dynamic whereas the execution order also referred to as scan-order is fixed.

Ferrarini *et al.* [9] sketch and compare different approaches for execution models that they categorize by the combination of two criteria: fixed or variable FB scan-order and the type of multitasking strategy used. Based on this

categorization they created test implementations for the assessment of different aspects of the timing behavior. They also reveal the problems with event connection loops caused by the execution model of the FBDK [14], which corresponds to the implementation approach “A0” (event propagation by direct call, no use of threads) in their catalog.

Zoitl *et al.* [10] provide a good insight into the vocabulary of concepts related to the different levels of execution elements in IEC 61499 and their scheduling. They discuss synchronization and preemption problems concerning the quasi-parallel execution of FBs that are connected to the same target FB. Furthermore, they propose an approach for timely execution of periodically triggered control algorithms with a given cycle time.

According to Thramboulidis [11], there were at the end of the year 2005 only two FB environments publicly available that support the generation of executable code from design specifications: the Archimedes System Platform and the well-known FBDK. The latter shows the problems explained in [9]. The former provided by Thramboulidis’s group is part of an elaborated framework and is described in detail in [12].

This situations encourages us to consider the adaptability of the handler-based execution model (HB-XM) that we defined for CNet for the following reasons: Although FBs as well as CNet describe control applications in an event-discrete manner, only [12] fully adopts the event-discrete execution concept for FBs, while [6] [7] focus on PLC-like systems and present a tool-based transformation into the scan-based execution paradigm. The HB-XM directly implements the event-discrete paradigm of CNet and since industrial control devices are more and more influenced by standard PC technologies, it can be assumed that future systems will provide the necessary features. Secondly, the HB-XM retains the independency of concurrent parts and finally, it forwards the partition and distribution of control applications to several devices by the transparency of location, which is easily achievable, because the HB-XM can be realized without any global data structures. The first parts of the HB-XM in the context of CNet were presented in [5].

The remainder of this paper is organized as follows. Section II gives an introduction of the main concepts of the HB-XM, followed by a class model to represent these concepts in Section III. Section IV shows the concrete mapping of IEC FBs elements to the HB-XM. Section V demonstrates an example, before the work is concluded in the last section.

## II. HANDLER-BASED EXECUTION MODEL

The handler-based execution model (HB-XM) forms an abstraction for the behavior of event-discrete modeling schemes that allows for automatic generation of efficient real-time code (Fig. 2).

Therefore the HB-XM defines the three basic abstract

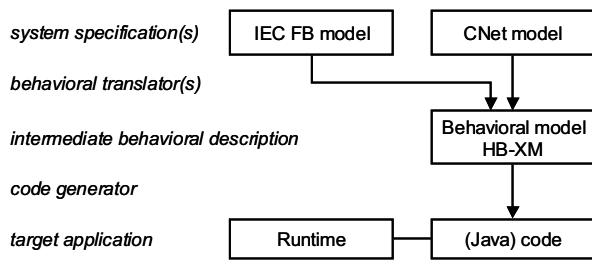


Fig. 2. HB-XM serving as an intermediate representation of event-discrete modeling schemes for automatic code generation

concepts *resource*, *event*, and *handler*. Their relations and interactions are suitable for describing the behavior of concurrent event-discrete systems and precise enough to automatically generate executable code thereof. *Event sets*, *timers*, and *monitors* are auxiliary concepts that are needed at runtime.

A *resource* shall be understood as an entity that describes a part of the system state at runtime and may be accessed and/or modified from two or more parts of the system. A place of a Petri net that is connected to at least two different transitions forms a *resource* in terms of this definition. (The use of the term *resource* in the context of the HB-XM must not be confused with a resource in terms of IEC 61499.) Resources provide operations to query and/or modify their state. Resources are called *passive* ones, if the state changes are exclusively effected by modify operations. *Active* ones may change their state autonomously, e.g. an activated timer fires without further request.

An *event* refers to a specific happening of negligible duration that indicates a state change in the system that might be of interest for some parts of the system, e.g. a modification of a resource or a fired timer. Events are simple, i.e. they do not carry ancillary data. Thus the receiver of an event cannot distinguish between events, if it receives the same specific event from different resources. Events can be bound to zero or more handlers.

A *handler* denotes a sequential software procedure that is divided into a condition part and an action part. A handler is activated and executed, if one of the events that it is assigned to occurs. The instructions in the action part are only executed, if the condition expressed by the condition part holds. The instructions may query resources in the condition part and they may query as well as modify

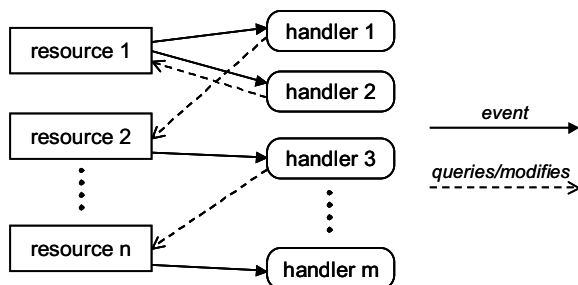


Fig. 3. Dependencies between resources and handlers

resources in the action part. Each modification may raise further events that activate subsequent handlers. However,

the activation may at the earliest be carried out, after the actual handler terminates. All events raised by the actual handler are considered to occur at the same time. If this has to be implemented on a quasi-parallel system, where a simultaneous activation is not possible, the actual activation order must have no influence on the correctness of the system. Fig. 3 shows how resources and handlers interact with each other. The assignment of handlers to certain events shall be called *binding*.

An *event set* assists in collecting raised events during the execution of a handler and activating the bound handlers after termination of the actual one. The implementation of the event set encapsulates the concrete scheduling strategies to guarantee fairness and enforce real-time constraints, if applicable.

*Timers* are special active resources that are needed to describe delayed event propagation, periodic event triggering, etc.

*Monitors* represent a means of inter-process communication to achieve mutual exclusion when accessing common resources.

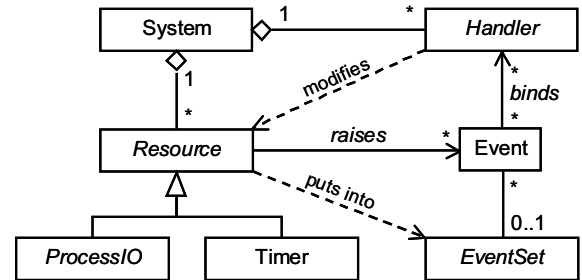


Fig. 4. UML class diagram for system models according to the HB-XM

Fig. 4 shows a UML class diagram representing the previously outlined concepts as classes with their associations and dependencies. One system is composed of several resources and handlers. A resource may raise events by adding them into an event set. Multiple emission of an event is not allowed. Each individual event is either included

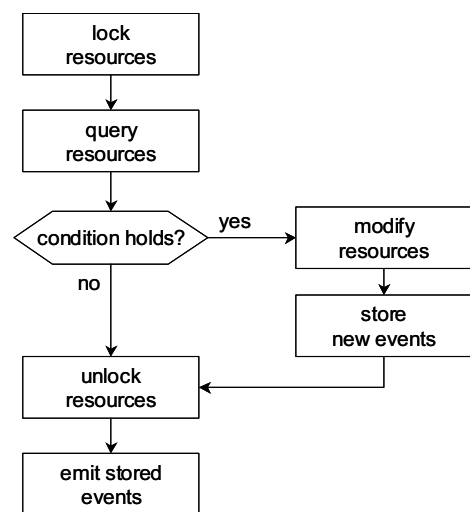


Fig. 5. Sequential structure of a handler implementation

in exactly one event set or not included in any event set. Each event can activate zero or more handlers, which on their part can modify the resources. Timers and process input/outputs are specialized cases of a resource.

### III. SOFTWARE REALIZATION ASPECTS

The previous section explained the elements that a system model to be represented by the HB-XM consists of. This section outlines implementation details that reflect the desired behavior.

#### A. Handler structure

A handler is split up into a condition that is evaluated depending on the state of resources and an action that is processed, if the condition holds. Since the handler can be executed in a multithreaded environment, where other thread possibly also access the resources the handler does, synchronization is required. Fig. 5 depicts the structure that is required for a handler implementation. Firstly, all used resources have to be locked. Secondly, the resources are queried so that the condition can be evaluated. When resources are modified, this may result in raising subsequent events. The propagation of these events to the system must be delayed till the locked resources are released. This is achieved by passing an event container object (see lines 10-11 in Fig. 6) to each resource modifying operation, which in turn stores new events in it.

Fig. 6 shows a code template that is used for the automatic generation of Java code from a handler. The parts

```

01 Handler h1 = new Handler() {
02     public void handle(EventSet es)
03     {
04         synchronized(r1) {
05             synchronized(r2) {
06                 int val1 = r1.getValue();
07                 int val2 = r2.getValue();
08                 if(val1 == 42 && val2 < 7) {
09                     int val3 = r1.getX();
10                     r2.performX(es, val2, val3);
11                     r1.performY(es);
12                 }
13             }
14         }
15     }
16 };

```

Fig. 6. Java code template for a handler implementation

in italics just serve as example for reading from resources (lines 6-7), evaluating a condition (line 8) and modifying resources (lines 9-11). Particular attention must be paid to the order of the resource allocation (lines 4-5). The example synchronizes first on r1 and then on r2. If a different handler that can be executed in the context of another thread tries to enter the resources' monitors in a different order, deadlock situations are possible. Potential deadlocks exist in multithreaded environments, if a directed cycle of inter process dependencies is possible [13]. Deadlocks are avoided, if such a cycle is made impossible. This can be

```

17 public abstract class EventSet {
18     public abstract void insert(Handler h);
19 }
20
21 class HandlerList extends EventSet {
22     private Handler first, last;
23     public void insert(Handler h) {
24         if(h.setParent(this))
25             if(first == null)
26                 first = last = h;
27             else {
28                 last.setNext(h);
29                 last = h;
30             }
31     }
32     public void run() { ... }
33 }

```

Fig. 7. Abstract event container class and concrete realization as list

achieved by defining a static order, in which resources are requested.

#### B. Event container realizes dispatching strategy

The EventSet class is abstract and forms the common base for different handler activation strategies. In Fig. 7 the definition and a possible implementation as list of handlers is given. Previous work [5] only offered this concrete class without the event-set abstraction that enables customized implementations.

#### C. Execution model

The dynamic behavior of the run() method (Fig. 7, line 32) called in order to invoke the collected handlers is explained in Fig. 8. The handlers that result from resolving an event are arranged as linked list (see Fig. 8(a)). When run() is called, the first handler ( $h_0$  in Fig. 8(b)) is uncoupled and the "first" pointer is adjusted, before the handler itself is invoked by calling its handle() method (cf. Fig. 6). When running, the action part of the handler may raise new events, whose bound handlers are appended to the end of the linked list (see Fig. 8(c)), if they are not already part of any list.

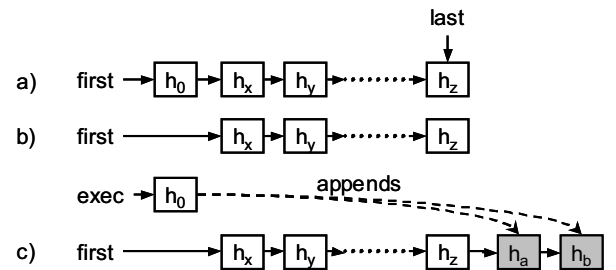


Fig. 8. Event container organized as linked list of activated handlers

This is possible via the reference to the list that is passed to the handle() call. After the call returns the procedure starts over again by uncoupling the next handler and so on.

This approach assures that (a) no handler is activated until the activating handler terminates, (b) calls are not nested, and (c) no recursion occurs even, if a handler reactivates itself. This means that the execution stack needed at runtime for the system is bounded and computable. If a handler is

activated twice, the second activation is ignored. This is fine, since activation means that the handler shall be executed, which is granted, since the first activation assures that the handler is already part of a list. This also guarantees an upper bound for the number of elements in any handler list.

#### D. Multi-threaded environments

The approach is transparent to multi-threaded environments provided that the synchronization to two or more resources is based on a total order as described above. Threads are a convenient means to concurrently observe process values and wait on system services. In general, threads should reside near any kind of event source and host a handler list, which serves as a container for the events it produces. This handler list is passed to the invoked handlers, so that subsequent activity is processed in context of thread until no following event is produced. This technique is even consistent to priority-based event sources. Since the handlers are processed in the context of the initial event, the event source's priority is promoted, so that subsequent handlers are processed under same priority.

#### E. Discussion

If one implemented the event set as shown in Fig. 9, the insertion of a new handler into the event set would

```

34 public class A0 extends EventSet {
35     public void insert(Handler h)
36     {
37         h.handle(this);
38     }
39 }

```

Fig. 9. Degenerated event propagation by direct call ('A0' approach)

degenerate to a direct call. This meant that subsequently issued events led to nested calls. If this was used as event set implementation in the following instead of the one previously described, the resulting behavior when executing the FB model could be compared to what Ferrarini [9] called 'A0 approach.'

### IV. MAPPING IEC-61499 FBs TO THE HB-XM

Although the HB-XM has been developed for CNet, the following will show that it can also be used for IEC 61499

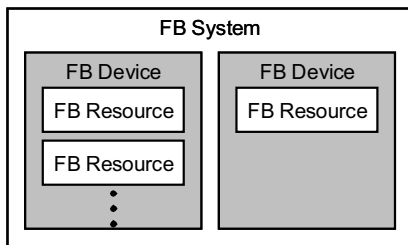


Fig. 10. IEC-61499 FB system with devices and resources

code generation. A system specification according to IEC 61499 consists of one or more devices. One device contains one or more FB resources as depicted in Fig. 10. A FB

resource hosts a network of different types of function blocks (FBs). Basic FBs directly encapsulate functionality, while composite FBs specify their functionality by means of a sub-network, which again consists of further FBs. This leads to a hierarchically structured network of FBs in each resource that can be described as a tree, whose leaves are exclusively formed by basic FBs. The reproduction of the behavior of a FB system is based on mapping basic FBs to the elements of the HB-XM. Two approaches are possible bottom-up and top-down. The former is translating the leaf FBs into the HB-XM and having a rule how to combine the resulting descriptions in the HB-XM domain according to the composite FBs. The latter means to transform the FB system first into flat model in the FB domain and translating it thereafter. In this work the latter approach is chosen.

#### A. Resolving composite FBs

In order to obtain a flat network all FBs that reside in composite FBs are recursively moved into the FB network

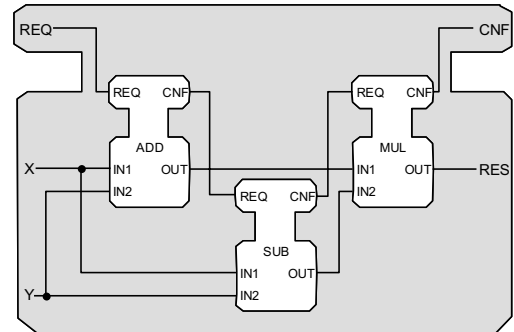


Fig. 11. Composite function block

that the composite FB is contained in. The interface of a composite FB is not transparent and cannot just be omitted, since it must buffer data values on event reception and for event propagation. For this purpose a new so-called latch FB is inserted to take over this functionality. The latch FB

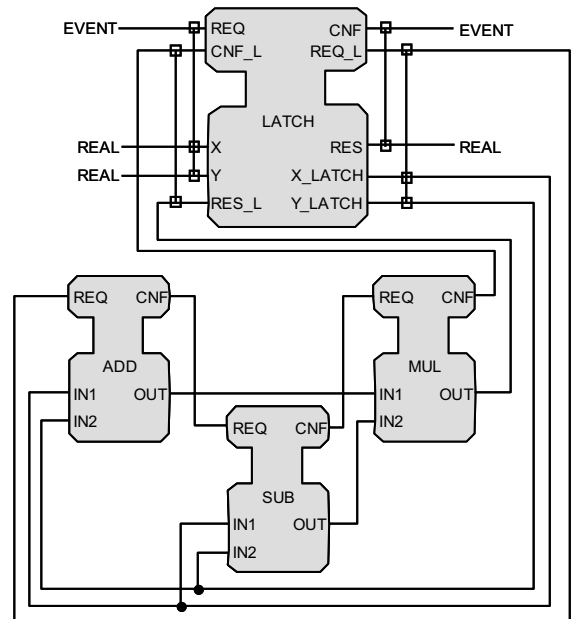


Fig. 12. Insertion of latch FB that replaces interface of composite FB

inherits the interface of the former composite FB extended in the way that for each event or data input a corresponding output is created and vice versa (e.g. X and X\_LATCH in Fig. 12). These form the interface to the FBs formerly located inside the composite FB. Fig. 12 shows the substitution for the composite FB in Fig. 11. Names of FB instances eventually have to be adjusted when moved in order to stay unambiguous (e.g. by prefixing with the former composite FB's name). The flattened network of FBs is transformed into the HB-XM FB by FB as described in the following subsection.

### B. Translating function blocks into the HB-XM

A basic FB is completely described by (1) its external interface, (2) an execution control chart (ECC) with (3) its algorithms, that can be implemented using different languages, e.g. structured text (ST), function block diagrams (FBS). The FBDK also provides Java.

#### 1) External interface

The external interface consists of event inputs and outputs, as well as data inputs and outputs. Each data input or output must be bound to at least one event input or output, respectively. Each event port associates zero or more

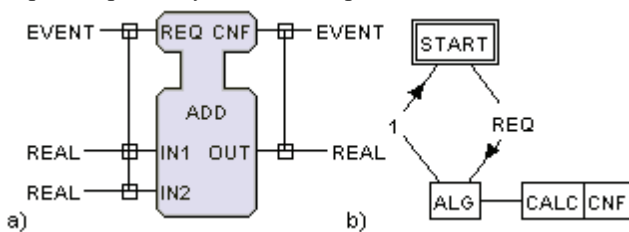


Fig. 13. a) External interface of FB with event and data ports b) ECC

data ports (depicted by a vertical line with squared connection markers, see Fig. 13). These bindings define which events are responsible for updating which data value. On the occurrence of an input event all data inputs associated with it have to be latched at the “inner side” of the FB interface in order to stay constant when processed by ECC and algorithms. The same is true for the outputs. In order to achieve this behavior through elements of the HB-XM the following elements have to be generated:

- All data ports are mapped to variables that represent (*passive*) resources. E.g. the interface of the FB in Fig. 13 requires a resource for IN1, IN2, and OUT.
- The latching operation is performed by *handlers*. For each event port that is associated with one or data ports a *copy handler* is needed. This handler is associated with the event corresponding to the event port. It synchronizes on the data connections and port resources and copies the values from the former ones

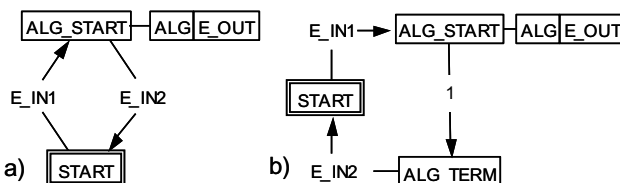


Fig. 14. a) Simple execution control chart with algorithm b) extended by a state that is entered unconditionally upon completion of algorithm

into the latter ones. Finally, it raises subsequent events for managing the execution control chart.

#### 2) Execution control chart (ECC) and EC states

The ECC defines a finite state machine consisting of states (called EC states) and transition arcs with Boolean expressions. These expressions may refer to event inputs, data inputs or both. The ECC in Fig. 14(a) has two states: START and ALG\_START. An EC state may be associated with an algorithm (e.g. ALG in Fig. 14) that is scheduled to be started upon entering the state and an event that is issued after the termination of the algorithm. The elements that have to be created for the HB-XM are as follows:

- For each ECC a variable is created as *passive resource* that stores which EC state is currently active. On each state change an *event* is issued.
- For each EC state associated with an algorithm an auxiliary state is inserted that is unconditionally entered upon completion of the algorithm according to Fig. 14(b). This simplifies the processing of event E\_IN2 which transfers the ECC back into state START only, if it is in state ALG\_START and the algorithm ALG has terminated.
- The termination of an algorithm will not directly issue the associated output event (E\_OUT) but an event activating a *copy handler* that is responsible for setting the output variables and queuing the output events. (This corresponds to the operations required at transition t4 in Table 1 of IEC 61499-1, also in [9].)
- A transition is translated into a handler evaluating the Boolean expression and eventually performing the state

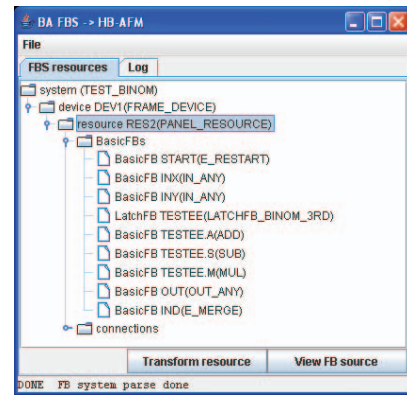


Fig. 15. IEC-to-HB-XM translator main window, FB system browser

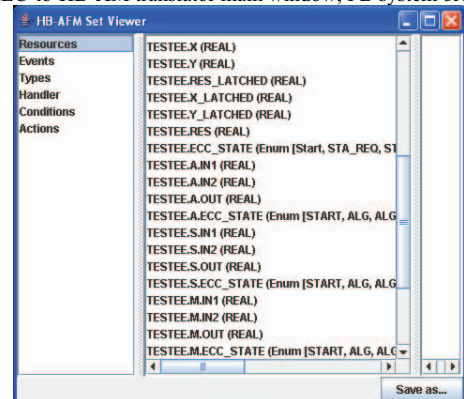


Fig. 16. HB-XM set viewer showing resources generated from composite function block from Fig. 11

change. If the Boolean expression contains a reference to an event input, the handler is bound to the corresponding event, otherwise to the event issued on entering the source EC state.

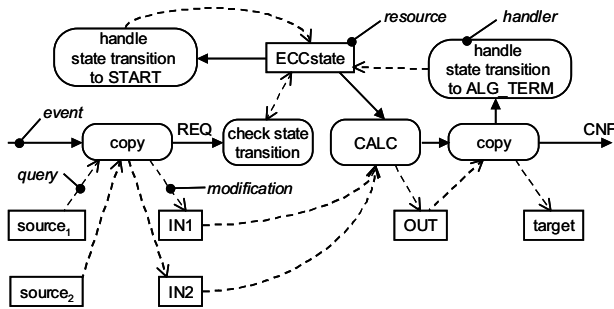


Fig. 17. Visualization of HB-XM elements generated for FB ADD

## V. EXAMPLE AND TOOL SUPPORT

The transformation of IEC 61499 function block descriptions into an HB-XM behavioral description as outlined in the previous section has been implemented in software (Fig. 15-16). The tool reads in FB system specifications created by the FBDK. The user can browse through the device-resource structure of the system (Fig. 15). Basic FBs or complete FB resources can be translated. The result is shown in a second window (Fig. 16) that shows the sets of generated HB-XM elements as lists. Fig. 17 visualizes the elements that are generated for the ADD FB that is part of the composite FB in Fig. 11. The synthetic latch FB (cf. Fig. 12) copies the data values into the data ports of FB ADD and raises the REQ event (Fig. 17), which activates the handler that manages the transition from EC state START to ALG (cf. Fig. 13(b)). Entering state ALG will activate the ECC algorithm CALC that appears as separate handler. At the moment, ECC algorithms must be directly provided in Java, since the translator does not include a parser for structured text (ST), yet. The termination of the algorithm triggers the copy handler to update the output data values and issue the CNF output event. The remaining handlers manage the trivial state transitions.

## VI. CONCLUSION

The handler-based execution model (HB-XM) has originally been developed in order to identify code patterns that match the event-discrete behavior of CNet to enable automatic code generation. In this paper we showed that it is a more general approach that is capable to describe the behavior of other event-discrete modeling languages, too. The paper demonstrated exemplarily its application to IEC 61499 function blocks. Furthermore, the HB-XM features the ability to enable concurrent event-propagation by restricting synchronization between different event paths to commonly accessed resources. All activity is based on the occurrence of events. HB-XM models are idle and do not consume any processing time, when no events occur. This is

in contrast to most known approaches that come from traditional PLC implementations that consist of loops performing the same operations over and over again, even if no input has changed. More modern approaches address this by trying to skip instructions that have no effect but are still based on a static scan-order. The HB-XM provides a full dynamic scheduling based on event propagation. Enforcing real-time constraints has not been addressed in this paper, but it is obvious that the code that the handlers consist of is either worst-case execution-time analyzable or restrictable to be so. Thus, deadlines are assignable to handlers, which have to be enforced by the underlying event set implementation. The concept of the event set is extensible for future needs, e.g. a real-time scheduler. Different event set implementation do not effect the code generation, since they are a part of the runtime library and all share the same interface (cf. Fig. 2).

## REFERENCES

- [1] H. Wurmus, "CNet—Komponentenbasierter Entwurf verteilter Steuerungssysteme mit Petri-Netzen," Ph.D. dissertation, Institute for Systems Engineering, Hannover, Germany, 2002.
- [2] N. Hagge, B. Wagner, "A New Function Block Modeling Language Based on Petri Nets for Automatic Code Generation," *IEEE Trans. on Industrial Informatics*, Vol. 1, No 4, pp. 226-237, Nov. 2005
- [3] G. Frey, "Automatic Implementation of Petri Net based Control Algorithms on PLC," in *Proceedings of the American Control Conference ACC 2000*, Chicago, June 28-30, 2000, pp. 2819-2823
- [4] N. Hagge and B. Wagner, "Mapping reusable control components to java language constructs," in *Proc. 2nd IEEE Int. Conf. Industrial Informatics INDIN'04*, Berlin, Germany, Jun. 2004, pp. 108-113.
- [5] N. Hagge and B. Wagner, "Java Code Patterns for Petri Net Based Behavioral Models," in *Proc. 3rd IEEE Int. Conf. Industrial Informatics INDIN'05*, Perth, Australia, Aug. 2005
- [6] J.L. Martinez Lastra, A. Lobov, L. Godinho, A. Nunes. *Function Blocks for Industrial-Process Measurement and Control Systems: IEC-61499 Introduction and Run-time Platforms*. Institute of Production Engineering. Tampere University of Technology. Report No. 67. ISBN 952-15-1244-X. Tampere, 2004
- [7] J.L. Martinez Lastra, A. Lobov, L. Godinho, "Closed Loop Control Using an IEC 61499 Application Generator for Scan-Based Controllers," in *Proc. 10th IEEE Int. Conf. Emerging Technologies and Factory Automation ETFA'05*, Catania, Italy, Sept. 2005
- [8] C. Jörns, L. Litz, and S. Bergold, "Automatische Erzeugung von SPS-Programmen auf der Basis von Petri-Netzen", in *Automatisierungstechnische Praxis - atp (37) 1995/3*, Oldenbourg-Verlag, pp. 10-14, 1995
- [9] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Proc. 2nd IEEE Int. Conf. Industrial Informatics INDIN'04*, Berlin, Germany, Jun. 2004, pp. 612-617
- [10] A. Zoitl, G. Grabmair, F. Auinger, C. Stünder, "Executing real-time constraint Control Applications modeled in IEC 61499 with respect to Dynamic Reconfiguration," in *Proc. 3rd IEEE Intl. Conf. Industrial Informatics INDIN'05*, Perth, Australia, Aug. 2005, pp. 62-67
- [11] K. Thramboulidis, "IEC 61499 in Factory Automation," in *Proc. Intl. Conf. Industrial Electronics, Technology and Automation*, Bridgeport, USA, Dec. 2005
- [12] G.S. Doukas, K.C. Thramboulidis, "A Real-Time Linux Execution Environment for Function-Block Based Distributed Control Applications," in *Proc. 3rd IEEE Intl. Conf. Industrial Informatics INDIN'05*, Perth, Australia, Aug. 2005
- [13] A.S. Tanenbaum, *Modern Operating Systems*, 2nd edition, Pearson 2001, ISBN 0130926418
- [14] FBDK, The Function Block Development Kit, HOLOBLOC, online: <http://www.holobloc.com/doc/fbdk/index.htm>