Modelling and Verification of IEC 61499 Applications using Prolog

Victor Dubinin
Penza State University,
Penza, Russia
victor_n_dubinin@yahoo.com

Valeriy Vyatkin
The University of Auckland,
Auckland, New Zealand
v.vyatkin@auckland.ac.nz

Hans-Michael Hanisch
Martin Luther University
of Halle-Wittenberg,
Halle, Germany
Hans-Michael.Hanisch@informatik.uni-halle.de

Abstract

This paper presents a new approach to modelling and verification of function block applications of the IEC 61499 standard. The approach uses the language of logic programming Prolog to represent a model of function block network and to verify its properties. The class of properties that can be checked is extended to more substantial queries providing in return not only "yes" or "no", but also the parameters explaining the reasons. The models essentially use the topological properties of the function block network and allow data of arbitrary types (not only Boolean) be used in the queries.

1. Introduction

In IEC61499 an application is represented as a network of *function blocks* (FB) interconnected by event and data connections [1]. A function block is an abstract component encapsulating some functionality that can be implemented by software or even by hardware. Some function blocks can be executed concurrently. The processes encapsulated in function blocks can communicate with each other thus increasing the complexity of the overall behaviour.

Concurrency and communication make considerable difficulties for behavioural analysis of function block systems. Traditional testing and simulation approaches are of limited applicability for many reasons. Recognizing that, a number of works appeared, for example [2,3,4,5,6], that suggest using formal modelling and verification to extend the analysis power. The mentioned works rely on modelling of function blocks with a state-transition modelling formalism, like Net Condition/Event Systems, Petri nets, Timed Automata, etc. The models are further analyzed on compliance with the *safety* or *liveness* properties using model-checking tools. For example, a typical safety property says that the system under control never gets to a dangerous state. A liveness property may assert that the system never deadlocks. However, the mentioned approaches are quite limited in providing answers to the questions like "at which values of parameter X parameter Y belongs to an interval *[a, b]*".

We have to mention that most of the modelling and verification approaches use open-loop models of the controller without a model of the behaviour of the controlled object. It is therefore questionable to verify liveness properties since they depend on the closed-loop behaviour of the controller together with the controlled system. In this paper we attempt to overcome these limitations by modelling of closed-loop systems of function blocks and by using the verification engine of Prolog language of logic programming, whose implementations contain a built-in deductive inference engine. Advantages of this modelling approach are simplicity, modularity and the ease of modifications.

The explicit event and data links between function blocks provide additional means for verification if compared with a general purpose software code.

We model function block systems using production rules [9]. A production rule has the general form of:

IF <condition> THEN <action>

This kind of models can be well implemented using Prolog [9] that is the most popular logic programming language based on the Horn's clause logic.

Prolog has several built-in mechanisms useful for verification purposes, e.g. the deductive inference mechanism with search and backtracking, and the mechanism of unification.

Prolog has been widely applied in various application areas. In particular, in [12] Prolog has been used for modelling and verification of discrete systems, presented in the form of high level Petri nets [10]. This modelling has some similarities with the function block model of IEC 61499 as it will be explained further in this paper.

Prolog queries allow formulation of complex specifications in terms of transitions and states, and, perhaps, even in terms of the application domain. Modal logics (such as temporal logic) can be applied on top of Prolog's inference engine for further improvement of the Prolog's power [11]. The verification process consists in the formulation of system properties in form of a Prolog query with subsequent evaluation of the query in the Prolog-based production system.

The verification process is inherently computationally complex. However, a useful selection of boundary constraints can help a lot in reducing the dimension of the space of reachable states. The constraints can be imposed as on the range of separate variables' values as well as on the combinations of values of variables, states and signals.

The constraints can be expressed in the form of Prolog clauses and be built directly to the interpreter of production rules implemented in Prolog. The main drawback of Prolog is its high computational demands (both in terms of memory and time), however these are getting compensated by the increasing computational power of available computers and increased efficiency of Prolog implementations. So, the use of Prolog won't add a qualitative raise in the resource requirements to the complexity of verification.

The rest of the paper is structured as follows. Section 2 presents a sketch of the function block semantic model. The model is presented in a rather informal way. It should be noted that the presented model has a more general nature than is required by the Prolog implementation discussed in this paper. The presented model specifies in more detail the semantic of function blocks as it is defined in the standard. In Section 3 the model is implemented using Prolog's artefacts, namely as a set of predicates and production rules. Interpretation of these rules is discussed in Section 4. Section 5 presents an illustrative example of a function block system modelling and verification using the developed model. Chapter 6 discusses the ways of properties' specification by the Prolog means. In conclusion (Section 7) the directions of future research are discussed.

2. The Function Block Model

1.1. Common information

In this paper we consider closed-loop networks of basic function blocks that correspond to the "Object-Control" or "Model-Control" design pattern, as simplified from the MVC pattern introduced in [7]. The considered networks are assumed to be "flat", that is: not include hierarchically other composite function blocks. Hierarchical structures of function blocks have to be "flattened". For that the composite blocks have to be substituted by their content appended by *data valves* implementing data transfer through their interfaces.

The idea of data valves is explained as follows. Composite function blocks consist of a network of function blocks and buffers for input/output variables. When translation of hierarchical composite blocks to a flat network is done, the data cannot just flow between the blocks of different hierarchical levels without taking into account the buffers. An illustration is provided in Figure 1.

The event/data associations of composite function blocks FB6 and FB7 (shown in Figure 1, upper part) determine sampling of the data while they are passed from block to block. As noted in [14], the event/data association can be arbitrary and not following the connections within the composite block, so they need special treatment when borders of the composite block are removed in the process of flattening.

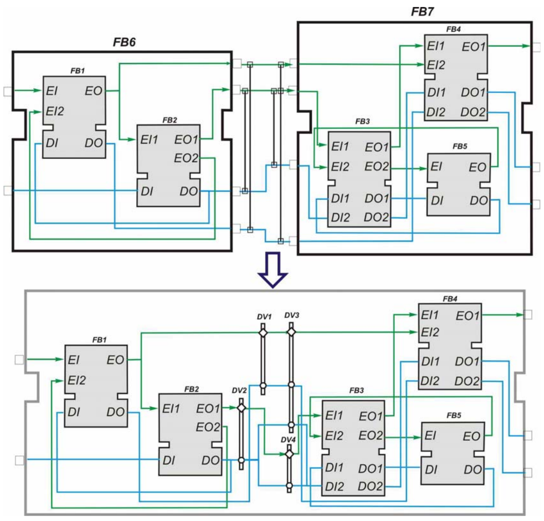


Figure 1. The function block obtained as a result of one step of 'flattening' with data valves.

For dealing with this problem, the concept of data valves with buffers was introduced in [13, 14]. The concept and notation of data valves are illustrated in Figure 2, a) and b) respectively.

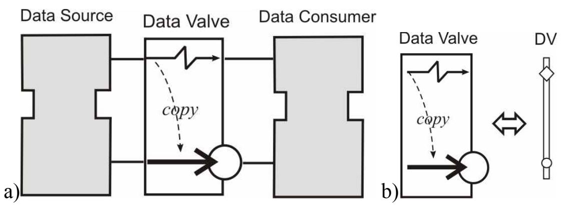


Figure 2. a) The data valve idea: the data input is copied to the data output of the valve when the event input arrives; b) compact notation of data valves.

Each outgoing and incoming event input (with their respective data associations) of a composite function block results in a data valve.

For the example presented in the upper part of Figure 1 the result of one step of the "flattening" with data valves implementing the "border issues" is presented in the lower part of the Figure 1. We do not represent the valves in the function block notation as we regard them to be a step towards a lower level implementation of function blocks. Also note with this respect that Figure 1 represents an operation called "merging of composite FBs" in the algebra of function blocks evolved by the authors in [13, 14].

A state of a flat function block network is determined by a tuple $S=(S^1,S^2,\dots,S^n)$, where S^i – is the state of *i*-th (basic) FB or data valve. As defined in [14], the state of the *i-th* FB is determined as $S^i=(cs^i, osm^i, ZEI^i, ZVI^i, ZVO^i, ZVV^i, ZBUF^i)$, where cs^i is a current state of *ECC* diagram, osm^i is a state of ECC operation state machine (ECC interpreter), ZEI^i – is a function indicating values of event inputs, ZVI^i, ZVO^i and ZVV^i – functions of values of input, output and internal variables correspondingly, $ZBUF^i$- function of data buffers' values (of unit capacity). The state of the *j*-th data valve is determined only by the function $ZBUF^j$.

One can note that the values of buffered data are included in the state of their respective function blocks or data valves instead of being directly included to the global network state. This is justified by the fact that a data buffer is associated with an output variable of function blocks.

In the following part of this Section we make some assumptions about the execution semantic of function blocks.

We are not specifically considering distributed configurations. Thus, modelling of *resources* and *devices* is beyond the scope of this paper.

The model presented in this paper uses non-deterministic selection of function blocks. This over-approximates the semantics of all scheduling policies in a particular FB system implementation. The non-deterministic selection allows exclude time from our consideration because this mechanism supposes examination of system functioning by all possible time parameters. However, a clear drawback of this method is redundancy, i.e. admission of scenarios that can never occur in a real system.

For the time being, we limit our consideration to "closed" networks of function blocks that do not receive events from the environment through the service interface function blocks (SIFB). Later on we show how the proposed model can be extended to cover the case of execution initiation from the environment.

This interpretation of the function block semantic is quite consistent and relies on the assumptions that a) function block is activated by an external event; b) execution of every algorithm is "short".

Although real interpreters of function blocks may have slightly different behaviour, the assumptions made above considerably reduce the number of intermediate states and determine the details of a legitimate implementation. Execution of a network of function blocks is activated by the *start event* that is issued only once. The start event leads to the action ***op6*** as described below.

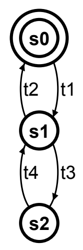
So, we can assume that a FB network transfers from state to state as a result of model transitions:

$$S_0[t_p\rightarrow S_1[t_q\rightarrow \dots [t_m\rightarrow S_n$$

Note that the proposed FB model can be combined with other state transition models such as Petri nets, NCES [2], etc. For this purpose it would be necessary to develop an interface for two kinds of models and rules of its functioning.

1.2. Semantic of function blocks

The IEC61499 standard partially defines the semantic of function block execution. In particular, the execution control of a basic function block is determined by Execution Control Chart, that is a kind of finite automata interpreted according to the ECC operation state machine shown in **Figure 3**.



State		Operations
s0		---
s1		Evaluate transitions
s2		Perform actions
Transition	**Condition**	**Operations**
t1	Invoke ECC	Sample inputs
t2	No transition clears	
t3	A transition clears	
t4	Actions completed	

Figure 3. ECC operation state machine [1].

The standard defines only some elements of the semantic leaving the details to the implementation. There is a number of works outlining the semantic loopholes of IEC 61499, in particular [14].

1.3. Types of model transitions

In the context of this paper, an ECC transition is said to be *primary* if its condition includes an event input (EI) variable. Otherwise, if it includes only a guard condition, it is said to be *secondary*.

The proposed model is a state-transition model. The model has five types of its state transitions (of the model, not of a function block's ECC!):

tran1	–firing of a primary EC transition;
tran2	–firing of a secondary EC transition;
tran3	–special processing of input event in an unreceptive state of FB;
tran4	–transition of the ECC interpreter to the initial state;
tran5	–working of a data valve;

The transition enabling rules are summarized in Table 1.

Type of transition	ECC interpreter state	Other conditions	Priority
Tran1	*Idle*	1) The source state of the EC transition is the current state of the (parent) function block; 2) The EC transition condition evaluates to TRUE;	3
Tran2	*Busy*		
Tran3	*Idle*	There is a signal at the event input (having WITH association(-s))	4
Tran4	*busy*	There are no enabled EC transitions	2
Tran5	*n/a*	This transition is enabled if there is a signal at the event input of the data valve	1 (highest)

Table 1. Conditions enabling the model transitions

The basic transitions determining the functioning of function block systems are the transitions of types 1 and 2. Transitions of the first type correspond to the almost complete cycle of ECC interpreter work (except the interpreter transition to the initial state *s0*), namely the chain $s0\rightarrow t1\rightarrow s1\rightarrow t3\rightarrow s2\rightarrow t4\rightarrow s1$.

Transitions of the second type represent the cycle $s1\rightarrow t3\rightarrow s2\rightarrow t4\rightarrow s1$.

Transition of the third type corresponds to the reaction on an incoming event and the corresponding sampling of the associated data variable in case when the ECC interpreter is *idle*, but the arrived event won't force any ECC transition. This type of transitions corresponds to the chain *s0→t1→s1→t2→s0*. Transition of the fourth type models transition of the ECC interpreter from state *s1* to the initial state *s0*. Transition of the type *tran5* models data sampling in a composite function block.

1.4. Compatibility and mutual exclusion of model transitions

Within the model of one function block some transitions are compatible (can be enabled simultaneously) and some are mutually exclusive. Based on the transition enabling rules introduced above we can build the relation of their compatibility/exclusion, presented in Table 2.

	tran1	***tran2***	***tran3***	***tran4***
tran1	+	-	+	-
tran2	-	+	-	-
tran3	+	-	+	-
tran4	-	-	-	-

Table 2. Table of model transitions' compatibility.

In Table 2 the "+" symbol designates that the transitions are compatible, while "-" shows that they are mutually exclusive. Thus, transitions of the *tran2* type are incompatible with *tran1* and *tran3* as they occur in mutually excluding states of the ECC interpreter. The *tran4* excludes any other transition by definition, and since data sampling in the "*busy*" interpreter state is impossible.

1.5. Firing transition selection rules

Firing transition selection rules define the order of enabled transition firing. Varying the firing transition selection rules it is possible to obtain different execute semantics of FBs.

The employed function block execution model combines the priority and the non-deterministic disciplines of active objects' selection from the set of enabled ones.

The hierarchy of priority levels is as follows. On the highest level is *the data valve* execution that has a higher priority (1) than *function block* since it is assumed that data valve's execution is by far shorter than a function block's execution.

At the function block level we introduce the following sublevels (in the priority descending order): 2) *tran4;* 3) *tran1* and *tran2*; 4) *tran3*.

A function block is said to be *enabled* if it has at least one enabled transition;

The selection of a firing transition is done by the following rules:

1. If there is a non-empty set of enabled data valves, one data valve is selected non-deterministically;
2. If there is a non-empty set of enabled FBs, then
   a. The FB to be current (i.e. active) is selected non-deterministically;
   b. Within the current FB, a transition is selected with the highest type priority and the highest priority within the type.

It should be noted that the priority of the third type transitions is determined by the priority of the corresponding EI-variable that, in turn, is determined by the location in the FB's textual representation (the earlier appears – the higher priority).

1.6. Transition firing rules

The *transition firing rules* define the operations executed at the transitions.

We define the following operations performed at the execution of function block systems.

op1 – Input data sampling resulting in a transfer of the data values to the corresponding input variables associated with the current event input by WITH declarations. In case of data valves the data is assigned to the external data buffer associated with the data valve.

op2 – Reset of all EI-variables of the current FB or data valve. This operation can be called "clearing the event channel" that eliminates the "event latching";

op3 - ECC interpreter jumps to the "busy" state;

op4 – Change of the current ECC state;

op5 – Algorithms' execution resulting in the modification of output and internal variables;

op6 – Transfer of signal(s) from event outputs of the current FB resulting in setting of EI-variables of the FBs and data valves connected to those event outputs by event connections. Prior to that the event channels of those recipient FBs are cleared to avoid "event latching". Alternatively, the events can be scheduled by sending the corresponding request to the event scheduler of the resource.

op7 – Transfer of output variable values (associated with currently issued output events) to the external data buffers.

op8 – Transition of the ECC interpreter to the "idle" state.

In Table 3 all model transitions are represented as sequences of some of the above defined operations (if the operation op_j, is a possible part of $tran_i$ then the corresponding table cell *(i,j)* is shaded.

	op1	***op2***	***op3***	***op4***	***op5***	***op6***	***op7***	***op8***
tran1	▨	▨	▨	▨	▨	▨	▨	
tran2		▨		▨	▨	▨	▨	
tran3	▨	▨						
tran4								▨
tran5	▨	▨						

Table 3. The model transition operation sequences.

Each action associated with a model transition is performed as a *transaction,* i.e. as an atomic non-interrupted action consisting in a sequence of operations executed in the pre-defined order.

In addition, to reduce the number of non-essential intermediate states it can be accepted that:

1) Transition of type 4 can be executed in a chain with transitions of type 1 or 2 as a single transaction;

2) Operation ***op6*** can be extended by including in it transmission of output signal from the FB-source to all FB-receivers through a network of data valves (if any)

including all data sampling operations in all involved data valves.

3. Declarative representation in Prolog

Below we consider representation of the model transitions using specialized production rules interpreted in Prolog. The start event is not represented by a rule, but just participates in formation of the initial state. The production representation reflects automata models, algorithms and connections between FBs. States of the networks of FBs can be effectively represented by means of tree structures which allow inclusion of semantic information along with structural information. Tree structures are well represented by means of Prolog.

The representation of a state as a Prolog's term has both benefits and drawbacks if compared to the traditional list-based representation. The benefits include simplification and homogeneity of the rules and of the rule interpreter, as well as higher effectiveness of unification of terms than that of the list processing procedures.

An obvious drawback is the dependency of the term-state dimension on the number of function blocks in the network, and on the number of their components.

The following mnemonic is used for the terms: *s*- global state of the FB network, *state* - state of ECC, *ei* - values of event input variables, *vi, vo, vv* - values of input, output and internal variables, *val* -value of an output variable; *buff* - value of the data buffer associated with output variable.

The partial syntax of the rule-based function block representation in Prolog is given below by means of EBNF (Extended Backus-Naur Form):

```
<production rule>::=
    tran(<type of model transition>,<identifier of FB-owner>, <transition
    name>,<previous state pattern>,<new state pattern>)
    [:- {<predicate of guard condition>[,<predicate of algorithm>]
  | <predicate of algorithm>}].
<type of model transition>::= ectran | sampling | freefb | datavalve
<previous state pattern>::=<state pattern>
<new state pattern>::=<state pattern>
<state pattern>::= s(<instance state pattern> [{,<instance state pattern>}...])
<instance state pattern>::= <basic function block instance state pattern>
  | <data valve instance state pattern>
<basic function block instance state pattern>::=
  < function block instance name>(state(<EC state identifier>),
  osm(<state of ECC interpreter>)
  [, ei(<list of event input variables>)]
  [, vi(<list of  input variables>)]
  [, vo(<list of output variables and data buffers >)]
  [, vv(<list of internal variables>)])
<data valve instance state pattern>::=
  <data valve instance name>(<event input variable>,bf(<list of data
buffers>))
<EC state identifier>::= <variable> | <symbolic constant>
<state of ECC interpreter>::= idle | work
<list of event input variables >::=
  <event input variable >[{,<event input variable >}...]
<event input variable>::= <event input variable's name>
  (<event input variable's value >)
<event input value >::= 0 | 1 | <variable>
<list of input variables >::= <input variable>[{,<input variable >}...]
<input variable >::= < input variable's name>(< input variable's value>)
<input variable's value>::= <variable> | <constant>
<list of output variables and data buffers >::=
  <output variable and data buffer >
  [{,<output variable and data buffer >}...]
< output variable and data buffer >::=
    <name of output variable>(val(<value of output variable>), buff(<value of
    data buffer))
<value of output variable>::= <variable> | <constant>
<list of data buffers>::= <data buffer>[{,<data buffer>}...]
<data buffer>::= <data buffer's name>(<data buffer's value>)
<data buffer's value>::=<variable> | <constant>
<variable>::= X<number of variable>
```

Note that the body of the Prolog clause *tran* is used only in case of model transitions of type *ectran*. The rules for model transitions of type *freefb* include only action but not enabling condition. The enabling condition is defined in rule interpreter for all transitions of type *freefb*.

The element <FB owner identifier> determines a unique alphanumeric identifier of an FB instance.

The Prolog representation of EC transition condition is built according to the following rule: the event-dependent part of the condition belongs to the pattern of the previous state in the Prolog clause header. The guard condition is represented in the Prolog clause body.

The algorithms are represented using predicates of the following general form:

<name of algorithm predicate> $(A_1, A_2,\ldots,A_n, B_1, B_2,\ldots, B_m)$,

where $A_1,A_2,\ldots,A_n$ – are input parameters of the algorithm, $B_1,B_2,\ldots,B_m$ – output parameters of the algorithm. Several Prolog clauses may be needed to represent an algorithm. Predicates of condition expressions and of algorithms are formed for function block types, not for instances.

4. Production systems operation

Functioning of a production system in many ways is based on the unification of terms which is used to determine resolvability of production rules. A production rule is enabled if 1) the term-pattern of previous state is unified with the term of the current state; 2) the predicate of the guard condition evaluates to true (if any).

The actions to determine new EC states, and new values of variables (event inputs as well as input, output, internal variables, and data buffers) in the term of the new state can be done in three different ways:

1. By explicitly stated new values. The new state pattern will have constants in the corresponding places;
2. Implementing algorithms as Prolog predicates. The predicate would compute the new values given the corresponding values of the algorithm parameters from the previous state. In this case the variables for old and new values shall be different;
3. Using inheritance of the state variables. This method is applied to the components that are not changed by the transition. Such components can be represented by the same variable in new and previous state patterns.

The work of the production system can be presented as a sequence of transactions.

An example of a simple interpreter of the production rules for function block systems' modelling is as follows:

```
path(S,[],S).
path(S,[IdT|L],Sn):- fbinstance(IdFB), priortran(Rtype,IdFB,IdT,S,S1),
Rtype\==freefb, releasefb(Rtype,IdFB,S1,S2), rstate(NS,S),
inbase(NS,IdT,S2), path(S2,L,Sn).
priortran(Rtype,IdFB,IdT,S,S1):- tran(Rtype,IdFB,IdT,S,S1),
Rtype\==freefb,!.
releasefb(Rtype,IdFB,S1,S2):- Rtype==ectran,
\+tran(ectran,IdFB,_,S1,_), tran(freefb,IdFB,_,S1,S2),!.
releasefb(Rtype,IdFB,S1,S2):- S2=S1.
inbase(NS,T,S1):-   rstate(NS1,S1),assertz(arc(NS,T,NS1)),!,fail.
inbase(NS,T,S1):-  retract(count(N)),NS1 is N+1, assert(count(NS1)),
assertz(rstate(NS1,S1)), assertz(arc(NS,T,NS1)).
build_RG:-
init(S),assertz(rstate(0,S)),assert(count(0)),!,path(S,_,_),fail.
```

In this version of the rule interpreter the priorities of FBs and data valves are considered to be equal.

The recursive predicate *path* is used to construct a chain of model transitions, transferring system from a source state to a target state. The database *fbinstance* stores the names (unique) of all FB and data valve instances containing in the FB system.

The predicate *priortran(Rtype,IdFB,IdT,S,S1)* is used to determine the state *S1*, immediately reachable from state *S*, as well to determine identifier *IdT* and type *Rtype* of the model transition that caused the state change. This change must be relevant to the function block (or data valve) *IdFB* and must have the highest priority in it. Priority of a production rule is determined by its location in Prolog program's text (the earlier rule is located – the higher is the priority).

The predicate *releasefb* is used to implement the model transition of type 4. It transfers the ECC interpreter to the state "*idle*", if the ECC has no more enabled EC transitions. The predicate *inbase* inserts information about a new state and a new arc of the reachability graph to the databases *rstate* and *arc*, respectively, and to determine if the new state duplicates an existing state. The generated states are numbered using a counter stored in the database *count*.

The predicate *build_RG* is used to construct the reachability graph from the initial state.

The presented interpreter can be used as a prototype for implementation in a more efficient way, for example, in C/C++.

5. Example

The following example serves to illustrate the analysis of a FB network containing all essential attributes, such as event input and output variables as well as input, output and internal variables. However, many aspects of function block systems are omitted for the sake of brevity.

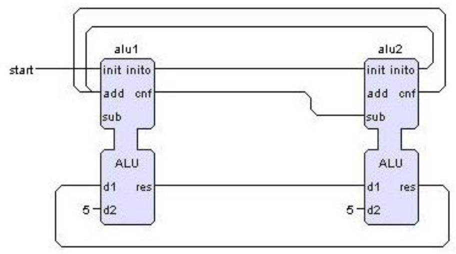


Figure 4. Function block network used for the verification trial.

The network, presented in Figure 1, consists of two instances *alu1* and *alu2* of function block type *ALU*. The outputs of one block are connected to the inputs of the other. After the initiating event the system must do the eternal sequence of actions, consisting of the interleaving addition and subtraction operations. The input parameters are selected in a way ensuring that the variable value limits are bounded. For example, if the first block adds some number, the second block would subtract the same number. This is intended to make the state space of the model better observable (for the purposes of this example).

The function block type ALU is shown in Figure 5. The block performs two arithmetic operations: addition *res=d1+d2+n*, initiated by the event input "add" and subtraction *res=d1-d2-n*, initiated by the event input "sub", where *n* – is an internal variable initialized with value 13. The output variable *res* is initialized with value 3.

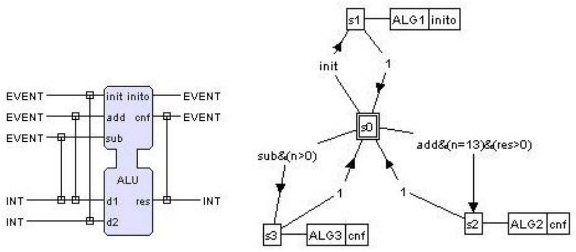


Figure 5. Description of the function block type ALU: interface and ECC.

The tree-structured global state of the function block net from **Figure** 4 is presented in Figure 6.

The following terms are used in the tree:

- *alu1* and *alu2*– states of function blocks *alu1 u alu2,* respectively;
- *init, add* and *sub* – values of the event input variables *init, add and sub,* correspondingly (0 or 1);
- *d1* and *d2*– values of input variables *d1* and *d2,* correspondingly;
- *res* – values of the output variable *res* (*val*) and its buffer (*buff*);
- *n* – value of internal variable *n.*

Leaves of the tree are numbered. The state pattern in the form of Prolog term is as follows.

```
s(alu1(state(X1),osm(X2),ei(init(X3),add(X4),sub(X5)),
vi(d1(X6),d2(X7)), vo(res(val(X8), buff(X9))), vv(n(X10)),
```

alu2(state(X11),osm(X12),ei(init(X13),add(X14),sub(X15)), vi(d1(X16),d2(X17)), vo(res(val(X18), buff(X19))), vv(n(X20))).

The initial state of the FB network is determined by the following term:

s(alu1(state(s0),osm(idle),ei(init(1),add(0),sub(0)), vi(d1(0),d2(0)), vo(res(val(0), buff(0))), vv(n(0))), alu2(state(s0),osm(idle),ei(init(0),add(0),sub(0)), vi(d1(0),d2(0)), vo(res(val(0), buff(0))), vv(n(0)))).

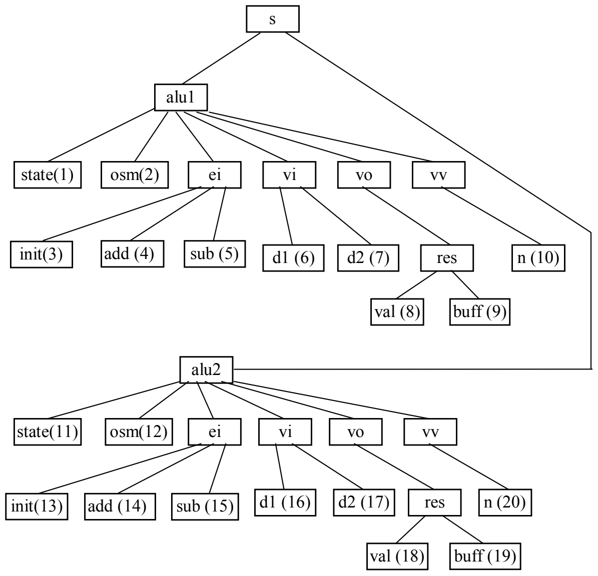


Figure 6. Tree-structured global state of the function block net.

Let us consider for example the transition *s0->s2* in the *ECC* of the block *alu1*. This model transition is of type "Primary EC transition".

The transition enabling conditions are :1)ECC of the block *alu1* is in the state *s0;* 2) event input signal *add* is present*; 3)* guard condition (n=13)&(res>0) is true*; 4)* ECC interpreter of block *alu1* is in state *idle*.

The actions performed by the transition are as follows:

1) ECC of the block *alu1* changes current state from *s0* to *s2;*
2) Reset all event inputs of the block*;*
3) Sampling the input variable *d1* associated with the event input *add* by a WITH construct. As a result of it, input variable *d1* of the block *alu1* is assigned with the value of the data buffer associated with the output variable *res* of the block *alu2;*
4) ECC interpreter of the block *alu1* goes from state *idle* to state *work*;
5) Output variable *res* is assigned the value *(d1+d2+n),* as determined by the algorithm *alg_alu_alg2;*
6) The output event *cnf* is initiated, and as a result set the event input variable *sub* of the block *alu2;*
7) The value of the output variable *res* is assigned to the data buffer associated with that variable.

The Prolog production rule representing this model transition is as follows:

```
tran(ectran,alu1,alu1_s0_s2,
s(alu1(state(s0), osm(idle), ei(init(0),add(1),sub(0)), vi(d1(X6),d2(X7)),
vo(res(val(X8), buff(X9))), vv(n(X10))),
alu2(state(X11), osm(X12), ei(init(X13),add(X14),sub(X15)),
vi(d1(X16),d2(X17)), vo(res(val(X18), buff(X19))), vv(n(X20)))),
s(alu1(state(s2), osm(work), ei(init(0),add(0),sub(0)),
vi(d1(X19),d2(X7)), vo(res(val(X8m), buff(X8m))), vv(n(X10))),
alu2(state(X11), osm(X12), ei(init(0),add(0),sub(1)),
vi(d1(X16),d2(X17)), vo(res(val(X18), buff(X19))), vv(n(X20))))):-
            cond_alu_s0_s2(X10,X8), alg_alu_alg2(X19,X7,X10,X8m).
cond_alu_s0_s2(P1,P2):-P1=13,P2>0.
alg_alu_alg2(A1,A2,A3,B1):- B1 is A1+A2+A3.
```

The reachability graph of the FB system considered above is shown in

Figure 7.

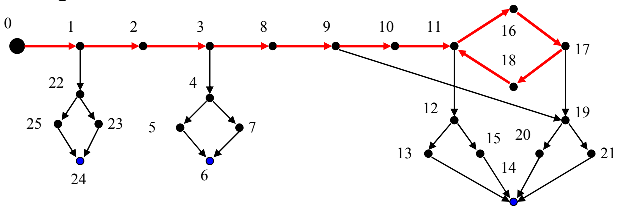


Figure 7 Reachability graph of the FB system.

As it can be seen from the graph, there are three deadlock states in the FB system. For example, a path to deadlock state 24 (including only EC transitions) is as follows: *alu1(s0→s1), alu2(s0→s1), alu1(s1→s0), alu2(s1→s0)*. The deadlock arises because of loss of a short-living signal on the event input *add* of block *alu1*. This signal is reset on the step *alu1(s1→s0)*. If the given FB network is executed on the one resource then it is possible to avoid the deadlock by means of a proper scheduling strategy. In case of multiple resources and devices it is necessary to take into account real time constraints and relations. The alternative way to avoid the deadlock is to restructure the FB system.

6. Specification of Properties

Prolog can efficiently represent various queries covering the reachability, liveness and safety of the function block applications. For example:

1) Determine, at which event input variable values the block *alu1* will be in the state (ECC) *s1*, and the *alu2* – in the state *s0*";
2) At which values of input variables *d1* both blocks will be in identical states?
3) Find states in which values of the variable *n* is greater than some constant A
4) "Are there any input events losses?", etc.

The properties are coded as Prolog predicates and make queries to prove or falsify.

The proof of the properties consists in automatic inference of a path to the state corresponding to a condition. The path is a sequence of production rules applied. It is automatically generated by the Prolog engine.

It is possible to specify FB system's properties using branching time temporal logic CTL (Computation tree logic) interpreted in Prolog [11]. For example, liveness of ECC transition t is expressed by the following CTL formula:

$$s_0 \models ALL(POT(enabled(t))).$$

where s_0 is an initial state, ALL and POT are temporal operators. The proposition *enabled*(t) is used to determine enableness of ECC transition t. The corresponding Prolog query is shown below:

```
?- init(S),all(pot(enabled(T)),S).
```

The interpreter of CTL formulas was developed in Prolog for using jointly with the rule interpreter. The Prolog program was implemented using SWI-Prolog [15].

To automate the generation of the Prolog representation of function blocks, a prototype converter was developed. The converter reads XML representation of function blocks and generates the Prolog production rules.

7. Conclusion

The paper reports on the first steps toward creating a powerful and flexible tool for analysis of function block systems. Further works will deal with the following issues:

1) The development of the corresponding intelligent GUI, supporting query formulation and interpretation of the results of the proof;
2) The development of structural constraints based on the description of incorrect situations in the graph form;
3) Adding models of service interface function blocks
4) Adding the concept of time to the modelling;
5) Use of semantic information and ontologies of application domains for verification;
6) Modelling of distributed function block configurations;
7) Reduction of the number of data valves and data buffers based on the traces of data flows, aiming at the reduction of the reachability space;
8) Experimental evaluation of resources use needed to execute the production system in various implementations of Prolog.
9) Implementation of the rule interpreter using conventional programming language, e.g. C/C++.

Special attention will be directed to deal with the integration of the developed methods and tools in the steps of the overall engineering process based on function blocks and special design patterns that enable us use closed-loop models of controllers and controlled systems.

8. References

1. *Function blocks for industrial-process measurement and control systems* - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
2. Vyatkin V., Hanisch H.-M. "A modelling approach for verification of IEC1499 function blocks using Net Condition/Event Systems", *Proc. IEEE conference on Emerging Technologies in Factory Automation (ETFA'99)*, Barcelona, Spain, 1999, pp. 261—270
3. H. Wurmus, B. Wagner, "IEC 61499 konforme Beschreibung verteilter Steuerungen mit Petri-Netzen", Conference Verteilte Automatisierung„ Proceedings, Magdeburg, 2000
4. Stanica P., Gueguen H., "Using Timed Automata for the Verification of IEC 61499 Applications", *IFAC Workshop on Discrete Event Systems (WODES'04)*, Reims, France, 2004
5. Faure J.M., Lesage J.J., Schnakenbourg C., *Towards IEC 61499 function blocks diagrams verification*, IEEE Int. Conference on Systems, Man and Cybernetics (SMC02), October 6-9, Hammamet, Tunisia, 2002
6. A. Lueder, C. Schwab, M. Tangermann, and J. Peschke. *Formal models for the verification of IEC 61499 function block based control applications,* IEEE Conference on Emerging Technologies and Factory Automation (ETFA'2005), Proceedings, Catania, Italy, September 2005.
7. Christensen J.H., IEC 61499 architecture, engineering, methodologies and software tools, 5th IFIP International Conference BASYS'02, Proceedings, Cancun, Mexico, 2002
8. Bonfe M., Fantuzzi C., *An Application of Object-Oriented Modeling Tools to Design the Logic Control System of a Packaging Machine*, Proc. 2nd International Conference on Industrial Informatics (INDIN'04), Berlin, Germany, 2004
9. Clocksin W.F., Mellish C.S., *Programming in Prolog*, 2nd edition, Berlin: Springer-Verlag, 1984.
10. Azema P., Juanole G., Sandus E., Moutbernard M. Specification and verification of distributed systems using Prolog interpreted Petri nets, Proc. 7th Int. Conf. Software Eng, 1984, pp.510 - 518
11. Papapanagiotakis G., Azema P., Pradin-Chezalviel B. Propositional branching time temporal logic in Prolog, Proc. 5th Annual Int. Phoenix. Conf. Comput. and Commun., 1986, pp.371 - 377
12. Dubinin V. N., Zinkin S.A. Logic programming languages for design of computer systems and networks, Penza State University Publishers, 1997, 88 p., available in electronic form at: http://alice.stup.ac.ru/~dvn/prolog/index.htm
13. V. Dubinin, V. Vyatkin, Formalized definition and modelling of IEC 61499 function block systems, Letters of Tertiary Education Institutions, Volga region, Russia, Penza State University Publishers, 2005, N 5, pp.76-8
14. V. Dubinin, V. Vyatkin, Towards A Formal Semantics of IEC 61499 Function Blocks, 4th IEEE Conference on Industrial Informatics (INDIN'2006), Singapore, 2006
15. SWI-Prolog web-site: http://www.swi-prolog.org/