On Controller and Plant Modeling for Model-based Formal Verification

Andrei Lobov, Jose L. Martinez Lastra, Reijo Tuokko
Institute of Production Engineering
Tampere University of Technology
P.O. Box 589, 33101 Tampere, Finland
{andrei.lobov, jose.lastra, reijo.tuokko}@tut.fi

Abstract

The acceptance of formal methods in industry is a challenging task mainly due to difficult learning process and the lack of the tools and methods helping control engineer to interpret the results of formal verification procedure. We model existing source code of the controller and controller related information (controller model) along with controlled object (plant model) and verify modeled system by means of model-checking.

This paper focuses on the issues related to controller and plant modeling. Ladder logic modeling is discussed along with the problems related to the plant modeling.

1. Introduction

We apply formal methods for modeling and verification of shop-floor equipment used in factory automation. The approach we utilize is the interpretation of existing source code and controller information to the model of the controller. The controller model is then interconnected to the plant model, which is derived from UML model. The closed-loop model is verified by means of model checking. Programmable Logic Controller (PLC) is seen here as a controller device used at the shop-floor.

Figure 1. Elements of verification procedure

As formalism for modeling, we use Timed Net Condition/Event Systems (TNCES), a Petri Net-based formalism. TNCES has been defined in [2]. We omit its details out of this paper. Here are given main definitions:

\[ TNCES = \{P, T, F, m_0, \psi, CN, EN, DC\} \] \[ \psi = \{C_0^\text{in}, E_0^\text{out}, C_0^\text{out}, E_0^\text{in}, Bc, Be, Cs, Dr\} \] \[ DC = \{DR, DL, Dc\} \]

Figure 1 demonstrates the approach: \( Tp \) is the function that translates UML diagrams to the TNCES [7]. \( Tc \) is the function that translates source code and controller information to the TNCES. Both functions are implemented in MOVIDA NCES Generator (MNG) software tool [6]. Thus the user of the MNG may generate the plant and controller models and interconnect them (C) using the same tool.

The outline of the paper is done as following: second section describes the LD instruction set which is translated into TNCES. Third section highlights plant modeling issues in relation to UML and analysis the role of the plant in model-based verification. Fourth section concludes the paper.

2. LD representation in TNCES

Modern controllers can be programmed in many different programming languages starting from low-level assembler-like languages up to high-level object-oriented languages (e.g. Java). Industrial programming languages and methods tend to be standardized. Most widely used standard is IEC61131-3 that describes four programming languages (Ladder Diagrams, Function Blocks, Instruction List, Structured Text) and one program organization unit (Sequential Function Chart). Despite of the standard, some of the PLC vendors holding a big share of the market keep there own implementation of some of the languages named in IEC61131-3. In our research, we have been considering mainly two programming languages: ladder diagrams (LD) (OMRON implementation) and Visual Flowchart Language (VFL) (Nematron implementation) [8]. MNG implements translation procedure of ladder logic into TNCES. The translation principles can be found in [6]. This section contains revised version of the translation procedure. There are a number of works done in PLC modeling [1], [4], [13], [14] – to name just a few. A good survey can be found in [15]. One of the added values of this work can be seen in the description of the non-trivial LD instructions modeling, e.g. CMP, BSET.
LD in OMRON implementation contains few hundreds of instructions. Development of the translator for all set of instructions is a challenging and sometimes unnecessary task. We have selected several instructions that were used in control logic of a case study example. The selected instruction set includes instructions that can compare two integer variables, set the memory areas to a certain value, serve as RS trigger, keep the previous scan cycle parameters and etc. The set may serve for wide-area control applications representation and can be seen as somewhat an instructions “basis”.

[5] gave general guidelines of modeling LD code with TNCES. There were changes made in the modeling of the controller and more detailed modeling instructions are listed below.

[5] described a skeleton of the PLC program, the revised version of the skeleton is shown in Figure 2. Skeleton is the initial model that is further enhanced while modeling LD instructions during translation of LD into TNCES.

Figure 2. PLC model skeleton. (p3, t3) contains scan cycle time interval

We define \( p1 \) as an input point of the skeleton. Transition \( t3 \) is said to form output points of skeleton. The translated racks of LD code follow \( t3 \) transition forming a new set of output points with each translated rack. Once translation is finished (‘END’ instruction is met in LD code) – the output points are interconnected with \( p1 \) place by means of the flow arcs. Input points consist of a single place – \( p1 \). Output points is a set of transitions which is updated with each translation of a LD rack.

The outcome of LD translation procedure is the controller model, which is composite TNCES (4). The skeleton is enhanced during translation procedure to form the logic execution model. Execution model derived from skeleton is one of TNCES modules in \( M \) (4). Besides logic execution model, the model of controller includes special and data TNCES modules. Special TNCES modules appear after modeling of certain LD instructions. Data TNCES modules represent Boolean variables modeling. During modeling phase we tend to represent any phenomenon state/parameter of the system by means of Boolean variables. The representation of Boolean modules in TNCES can be found in [5], [6], [8].

Let’s consider the modeling of the following LD instructions or elements (in parentheses is given a mnemonic of the instructions):

1. **Coil** – sets to TRUE (ON) or FALSE (OFF) associated variable.
2. **Timer** (TIM) – Counts a predefined time interval and sets the designated bit ON once the time has elapsed.
3. **Differentiate Up** (DIFU) – Checks if execution condition is ON during current scan cycle and it was OFF at the previous scan, the designated bit will be turned on. If the execution condition of the instruction was ON at the previous cycle and at the current cycle the execution condition is ON or OFF – the designated bit will be reset.
4. **Differentiate Down** (DIFD) – Checks if the instruction execution condition was OFF or ON in the previous cycle and became ON at the current cycle – the designated bit is turned OFF. If the execution condition of the function has been ON at the previous cycle and became OFF in the current cycle the designated bit is set ON.
5. **Reset** (RSET) – Resets the designated bit.
6. **Set** (SET) – Sets the designated bit.
7. **Keep** (KEEP) – Performs as RS trigger. Has two inputs S and R. If S input is ON the designated bit is set. If R input is ON the designated bit is reset.
8. **Compare** (CMP) – Compares two integers. The result may be: first operand is less than second, operands are equal, first operand is greater than second.
9. **Block Set** (BSET) – Initializes given memory area to the particular value.

Figure 3 shows an initial controller model TNCES\_composite (4), which includes two TNCES modules (1): change\_check and Skeleton.

Figure 3. Initial controller model

The change\_check module has been described in [5]. Its function is to monitor if any change occur somewhere in the Net. All the model elements, which should report the change in the state (transition of the token), has to be connected to the change event input of change\_check module or to the change\_in\_Plant event input of controller model in the case of external components for the controller model.
Ladder Diagram program (LD) is composed of finite number of racks, where each rack (R) can be seen as a triple:

\[ R = \{C, I, \varphi\} \] (5)

where \( C = \{c_1, c_2, ..., c_k\} \) is a finite set of Boolean conditions formed by LD contacts, \( I = \{i_1, i_2, ..., i_m\} \) is a finite set of instructions and coils in the rack and \( \varphi : i_j \rightarrow c_i \) is a function that maps instruction \( i_j \) of \( I \) to condition \( c_i \) of \( C \).

General algorithm of LD translation to TNCES may look as follow:

**Algorithm 1: LD translation to TNCES**

**Input:** LD source code and TNCES composite (Initial controller model - Fig. 3)  
**Output:** Complete controller model representing execution logic provided by LD program – TNCES’ composite.

Initialize Input Points in Skeleton (TNCES skeleton) \( inp \) to \( p_1 \)
Initialize Output Points in Skeleton (TNCES skeleton) \( out \) to \( t_3 \)

TNCES logic := TNCES skeleton

For each rack \( R \) in LD

IF ‘END’ \( \notin I \)

Create place \( p \) in TNCES logic
Interconnect transitions in \( out \) with \( p \): \( I \times p \)
Let \( out' \) be the output points for the parsed rack \( R \)
\( out' := \emptyset \)

For each condition \( C \) in LD
Parse Boolean expression of \( C \)
Define Transitions Set \( T \)
\( T_{outout} := \emptyset \)
Define Condition Inputs Set \( CI \) for \( C \)
For each contact \( cont \) in \( C \)
Define Boolean input TNCES module \( TNCES_{bi} \)
\( M := M \cup TNCES_{bi} \), where
\( M \in TNCES\_\text{composite} \)
Interconnect \( TNCES_{bi} \) with \( c_i \) of \( CI \)
Interconnect \( TNCES_{bi} \) with \( TNCES\_\text{change\_check} \)
Add \( CI \) to TNCES logic: \( C^m := C^m \cup CI \)

For each instruction \( i \) in \( I \)
Define condition \( c, c := \varphi(i) \)
Find transition set \( T' \) obtained when parsing condition \( c \)
Parse \( i \)
Define event outputs set \( EO \) in TNCES logic to trigger \( i \)
Interconnect \( T' \) and \( EO \): \( T' \times EO \)
Define special modules \( TNCES_{\text{special}} \) or Boolean outputs modules \( TNCES_{bo} \)
\( M := M \cup (TNCES_{bo} \mid TNCES_{\text{special}}) \)**
\( out := out' \)

ELSE ‘END’ \( \in I \)
Interconnect transitions in \( out \) with \( inp \)
\( TNCES\_\text{composite} := TNCES\_\text{composite} \)
For each \( TNCES_{bi} \) and \( TNCES_{bo} \) define inputs and outputs in \( TNCES\_\text{composite} \)

\[ M := M \cup TNCES_{\text{logic}} \]

Return TNCES’ composite

Described algorithm defines general steps in translating LD to TNCES. It omits some details, for instance, how to obtain transitions set \( T \) out of Boolean expression, condition inputs set \( CI \) or event outputs set \( EO \) or how to parse each instruction. We describe here in details LD instructions translation process**. The LD instructions were listed before. The translation algorithm is implemented in MOVIDA NCES Generator (MNG) software tool.

**2.1. LD Coil**

LD Coil turns designated bit ON if condition expression \( c \) is evaluated as TRUE, otherwise the designated bit is turned OFF. Designated bit is a Boolean variable, thus corresponding \( TNCES_{bo} \) is generated and added to the controller model - TNCES’ composite.

![Figure 4. LD Coil in TNCES](image)

Figure 4 shows extension of TNCES skeleton (at the bottom) when modeling LD rack (at the top). The numbering of places and transitions is done accordingly to the location of the parsed LD rack. Parsing the LD rack shown in Figure 4 should produce \( TNCES_{bi} \) module corresponding to ‘IX_Tank_Full’ contact and \( TNCES_{bo} \) module corresponding to ‘QX_Valve’ coil.

**2.2. Timer (TIM)**

Timer instruction requires a special TNCES module \( TNCES_{\text{special}} \) and \( TNCES_{bo} \) to be added to controller model. \( TNCES_{\text{special}} \) is a counting module for the timer. \( TNCES_{bo} \) stores the result of the timer count (time elapsed/not elapsed). \( TNCES_{\text{special}} \) is shown in Figure 5.

![Figure 5. Timer module (TNCES_{special})](image)

**2.3. Differentiate Up (DIFU)**

This LD instruction checks if the execution condition \( c \) became ON at the current scan cycle then it turns ON
designated bit. In other cases the bit is turned OFF. The instruction is modeled with the TNCES\textsubscript{special} shown in Figure 6 interconnected with TNCES\textsubscript{logic} on one side (DIFU\_ON, DIFU\_OFF) and TNCES\textsubscript{bo} on the other (Set\_var, Reset\_var).

![Figure 6. DIFU module (TNCES\textsubscript{special})](image)

2.4. Differentiate Down (DIFD)

‘Differentiate Down’ instruction monitors its execution condition (c). If c becomes OFF at the current scan cycle - the designated bit is turned ON. In all the other cases – the designated bit is turned OFF. Figure 7 demonstrates TNCES\textsubscript{special} that is added to TNCES\textsubscript{composite} when translating DIFD instruction.

![Figure 7. DIFD module (TNCES\textsubscript{special})](image)

TNCES\textsubscript{composite} also gets TNCES\textsubscript{bo} corresponding to the designated bit of the instruction.

2.5. Reset (RSET)

‘Reset’ instruction resets the designated bit. It can be compared to the coil modeling described in section 2.1. The difference is that the coil may affect its designated bit no matter if the condition of the rack is evaluated to TRUE or FALSE. The RSET instruction resets its designated bit iff the condition is evaluated to TRUE. Considering Figure 4, if the coil would be replaced with RSET instruction with the same designated bit ‘QX\_Valve’, then the difference in the modeling of the rack is that (t2, Set_QX\_Value) event arc should be removed. TNCES\textsubscript{bo} representing ‘QX\_Valve’ and TNCES\textsubscript{bi} representing ‘IX\_Tank\_Full’ tags have to be also added in the case of RSET modeling.

2.6. Set (SET)

‘Set’ instruction is used to set the designated bit. If the execution condition of the instruction is evaluated to TRUE the designated bit is turned ON. Again, considering Figure 4, if the coil would be replaced with SET instruction with ‘QX\_Valve’ as a designated bit, then the (t1, Reset_QX\_Value) event arc should be removed and condition input labels have to be interchanged to represent the TNCES\textsubscript{logic} extension for modeling SET instruction. The data modules representing LD contacts and designated bit are added to the controller model - TNCES\textsubscript{composite}.

2.7. Keep (KEEP)

KEEP instruction acts as RS trigger. It has S and R inputs. Once the execution condition of S c\_s is ON, the designated bit is set. If the execution condition of R c\_r is OFF the designated bit is reset. (Having both S and R ON, the designated bit is reset).

LD rack that contain KEEP instruction, which has two input execution conditions c\_s and c\_r acting on the same output, can be represented as two LD racks where c\_s is execution condition of a SET instruction and c\_r is an execution condition of a RSET instruction.

![Figure 8. KEEP instruction modeling](image)

Figure 8 demonstrates KEEP instruction modeling principles. TNCES\textsubscript{logic} enhancement is shown below the modeled LD rack. Place p1, transitions t1 and t2 and their incoming and outgoing arcs in Figure 8 correspond to modeling of SET instruction, while place p2, transitions t3 and t4 and their incoming and outgoing arcs form RSET instruction model. Combined together, we obtain KEEP instruction representation in TNCES. There are two TNCES\textsubscript{bi} (0.01, 0.02) and one TNCES\textsubscript{bo} are added to TNCES\textsubscript{composite} during translation procedure of a given example.

2.8. Compare (CMP)

CMP instruction compares two integer operands. Although the representation of the non-Boolean data is problematic in TNCES, there is trade-off available in modeling of the instruction. The state-explosion problem that may be caused by trying to cope with integers in this case can be narrowed to the modeling of the comparison
operation results. There are three possible outcomes of the comparison operation: first operand is greater than the second one; operands are equal; or the first operand is less than second. Thus, we need only three Boolean variables $TNCES_{\text{bi}}$, to represent the results of the CMP instruction.

Furthermore, we may not need all the results. For instance, the control program may just monitor when a certain parameter does not exceed a predefined value. In this case it may be enough to use only one Boolean variable that would be TRUE when the parameter is in the operational range.

When modeling CMP instruction, maximum three $TNCES_{\text{bo}}$ modules are added to $TNCES_{\text{composite}}$. Each module corresponds to GREATER, LESS or EQUAL result of the operation respectively. The status update of the modules is entirely plant task, since it is usual that the evaluation of controlled process answer the question if a certain controlled parameter is in particular value range. Therefore $TNCES_{\text{logic}}$ is not modified when CMP instruction is met.

There can be the cases when this approach won’t work. For example, if there is a counter that constantly increases certain integer in the LD logic program each scan cycle. In such cases it is necessary to look where and how this data is later applied. If the data is fed to the plant, it may be possible to represent such a behavior in plant model. In many cases all the integer parameters sooner or later compared to some threshold values, this is a right point to investigate if the outcome of the comparison can be used instead of representing the values of integers.

2.9. Block Set (BSET)

Another non-trivial instruction is a BSET. It inputs memory area (starting word – end word) and initialization value (source word). The memory area is filled with the source word.

The modeling of such instruction is relatively easy: the memory area is analyzed for stored variables. All the variables located in the modified memory area are identified. Depending if a variable is input or output $TNCES_{\text{bo}}$ or $TNCES_{\text{ba}}$ are created and added to $TNCES_{\text{composite}}$.

Event output set $EO$ (Algorithm 1) is interconnected with $TNCES_{\text{bo}}$ and $TNCES_{\text{ba}}$ modules to affect the values of these variables when the instruction is executed.

3. Plant modeling, Role of the Plant and Verification procedure

Model of the plant can be seen as a stub for the model of the controller. It bears the knowledge of controller scan-cycle outputs processing and generation of inputs values to the controller. Plant model may replace input test sequences that may be generated for the controller verification in the case of single controller model verification. Separation of plant and controller modeling processes allows model developers to concentrate specifically on plant model development. The final goal here is not to develop the plant model for the controller model, but to model the plant without taking into account specific features of the controller – to create independent plant to avoid error propagation done in controller to the model of the plant. The minor changes to the plant and controller models still may be required during models interconnection phase (C in Figure 1) when the I/Os of the plant model are mapped with controller I/Os.

As it was described in previous sections, the controller model is generated out of the source code and controller specific information. For plant modeling we apply Unified Modeling Language (UML) [16]. UML has gained a wide-acceptance in industrial world, especially in software engineering industries. A number of research works has been done in application of UML in design, modeling and analysis of industrial/manufacturing systems: [9], [10], [11], [12], [17] - to name just a few. The UML has been applied in different areas and fields in industry for design ([9], [11]) and analysis ([10], [12], [17]).

UML has a dozen of diagrams that allow to represent different aspects of the modeled system. In the beginning [8], [6], we model the plant directly in TNCES, which was time consuming and error-prone process. We decide to utilize a higher-level language such as UML for modeling of the plant [7]. Despite the fact that UML can be considered as a higher-level language compare to TNCES, its expression power allows to model the same phenomena as the TNCES does. One of the advantages in UML is that its diagrams allow modeler to concentrate on a specific feature of the plant. We identified three main aspects that has to be modeled [7], these are:

- Structure/interface of plant components
- Behavior of plant components
- Interconnections of plant components and components themselves - objects

To address abovementioned points, three UML diagrams are currently employed; these are respectively class diagrams, state charts and collaboration diagrams.

The automatic translation procedure of UML models composed of the selected diagrams is described in [7] and implemented in MNG.

Another UML diagrams are considered to be useful not for the plant modeling but for the requirements specification. These are use case diagrams and sequence diagrams. Use cases can be applied for generation of system requirements. The information that can be obtained from the diagram is in natural language. Therefore there is a need to define certain rules for constructing use cases to automate the process of requirements generation. However, it is probably one of the main challenges to give as much freedom as possible to the modeler on one hand and try to automate plant model generation process in order to ease a life of a human modeler on the other hand. Here is again a trade-off problem between the automated model generation and user-friendly modeling.

Sequence diagrams have more strict syntax and it is a good candidate for expressing system requirements. Sequence diagram shows particular scenario of system evaluation, which can be searched in the entire model of interconnected controller and plant models.
Is it possible to verify system without plant model (considering model-based approach)? It is applicable if the plant does not produce any delay between the input from the controller and the output to the controller. An example of such case could be a system where the controller produces output on time basis without taking into account the “feedback” from the plant – a traffic lights can be seen as an example of such system. We may construct the traffic lights program that would change the lights in turn keeping the possible combinations for predefined time period. Even if we decide to provide a feedback from the lights telling if the certain combination has been actually turned ON or OFF, it is still possible to avoid plant model by directly connecting outputs of the controller to its corresponding inputs. Nevertheless, plant modeling allows to represent non-standard situations.

The plant model is needed when the output of the plant is a function of time. The trivial example that requires plant model was demonstrated in [5]. A simple tank that filled with a liquid when input valve is open is one of the examples that requires plant model.

[Figure 9. Tank Plant model [5]]

Figure 9 shows the plant model of the tank (at the top – in IEC61499 notation) and the simple process description (at the bottom). The tank is filled if valve is open for at least 10000 time units. To control such a plant it is enough “one”-rack LD program shown in Figure 4 (at the top).

Besides the situation when plant output is a function of time, there may be controller specific features that sometimes can be thought of as a plant related. For instance OMRON CPM1A PLC has a ‘first scan flag’ that is set once the controller is in run mode and at its first scan-cycle. On the other hand the power of the controller can be considered as a plant specific feature (user turns ON/OFF a system). Thus, it can be seen that this flag has to be ‘controlled’ from the plant rather than in controller. On the other hand plant model has to be free of the controller specific features.

System verification is done by means of model checking. We utilize integrated Model Assembler and CHecker (iMATCH) [18] and SESA model-checker [19].

Checking of CTL formulae. Both tools generate a reachability graph of the system where the search of specific states or sequences of the states is carried out. As an analysis of the results the iMATCH produces time diagrams for the selected place markings. With the time diagrams, user may trace a specific scenario. One of the problems exhibited by application of CTL formulae in result analysis is that these allow to express sequence of the states but not the time between the particular two states. The later can be done with the timing diagrams. Thus, the verification process looks as follows, the user checks with logical prepositions and CTL formulae the availability of a particular scenario in principle and has to browse the timing diagrams to clarify the scenario. An importance of timing diagrams can be demonstrated with the following example:

[Figure 10. Bi-directional conveyor]

Figure 10 shows a bi-directional conveyor. Symbols ‘s’ mark the location of the sensors that can detect the pallet sliding on the conveyor belts, ‘M’ is a motor that may run clock-wise or counterclockwise. Lets consider the application where the pallet should constantly move between two sensors. Lets the pallet initially would be located on the leftmost sensor in Figure 10. The development of a control program may end with the program with the following logic:

1. Run motor to move pallet to rightmost sensor direction. Keep running the motor while the right sensor is not reached.
2. Once right sensor is reached: Turn off the motor, change the direction of the motor, run motor to left sensor direction. Keep running the motor while the left sensor is not reached.
3. Once left sensor is reached: Turn off the motor. Go to step 1.

Being in the world of models one could easily develop the logic described by these three steps. Verification of this logic would produce an expected result – the pallet will be moved between the sensors. Uploading such a code to the real controller may cause real motor to burn. The reason is that usually the motors cannot handle the sudden change of the direction. One could blame the requirements definition (1-3) as a cause of the problem. There is no time delay specified between the changes of the motor run direction. Or should it be plant modeling problem? This situation is quite possible that in modeled world human perception of the potential threads is weakened. Logical prepositions and CTL formulae verification may exhibit a correct behavior: there is a pallet on left sensor after being on right sensor and there is a pallet on the right sensor after being on the left sensor. The timing diagrams are the right tool to verify...
the time delays between the changes of system critical parameters.

If we consider plant modeling of a given example (Fig. 10) with UML, one of the main challenges is to represent discretization of the pallet positions on the conveyor belt. Currently it is done with the sequence diagrams: the conveyor is “split” into several logical positions and each position is represented via state in state chart.

Figure 11. State chart of the bi-directional conveyor

Figure 11 shows the UML state chart for the bi-directional conveyor (Fig. 10). There are seven states that model the positions of the pallet on the conveyor and two states representing the situation when the pallet has left the conveyor segment either at the left side or at the right side.

Class diagram for the bi-directional conveyor composed of a single class is shown in Figure 12. There are two inputs: motor and direction and two outputs: sensor1 (left) and sensor 2 (right). The set of methods in the class allows to manipulate its attributes. The manipulation logic is implemented in state chart (Fig. 11). Current syntax for plant model development with UML for automated translation to the TNCES is described in [7]. As can be seen in Figure 11, discretization problem involves representation of the each position of the pallet as a state in state chart.

Currently we consider revising the allowed UML syntax [7] introducing position or level modeling via class attributes. Figure 13 shows similar ‘Position’ class as it is depicted in Figure 12. The only difference is the addition of private pos attribute that should store the current position of the pallet. Thus, the state chart (Fig. 14) can be simplified to the three basic states.

Class diagram for the bi-directional conveyor position

Figure 13. Elaborated class diagram for bi-directional conveyor

Figure 14. Elaborated state diagram of bi-directional conveyor

4. Conclusion and future work

The modeling of the controller and the plant is often the search for the trade-off in representation of certain element of programming language or the phenomenon of the plant. Most relevant of those at this moment were discussed in the paper. The paper described modeling principles of “non-trivial” LD instructions that may operate with integer operands or with the memory areas.

In the future developments we are planning to extend automatic translation capabilities to some of Siemens S7 programming languages. Another important area is development of the methods and tools that would allow factory-floor engineer to interpret the outcome of model-checking results. Also the UML diagrams involved in system modeling and analysis are due to be extended allowing representation of system requirements.

5. Acknowledgement

This research is supported by Finnish National Technology Agency (TEKES) and industrial consortium: CIMCORP Oy, FlexLink Automation Oy, Metso Panelboards Oy, Omron Electronics Oy and PMJ Automec Oy.
References


