FORMAL SPECIFICATION OF PLC PROGRAMS USING TEMPORAL LOGIC

GIRISH KESHA PALSHIKAR and KESAV V. NORI
Tata Research Development And Design Centre (TRDDC),
1, Mangaldas Road,
Pune 411050, INDIA
email: {pgirish, kvn}@trddc.ernet.in

ABSTRACT

Programmable Logic Controllers (PLC) form the backbone of automation in discrete manufacturing factories, power plants and other critical systems. Most PLC programming techniques do not facilitate program verification, modular development, code generation, easy maintenance etc. There is also a need to reduce the resources spent in development and maintenance of large PLC programs. To meet the stringent requirements of safety and performance of PLC programs and to facilitate hardware-independent logical modeling of the application system, we advocate the use of advanced software engineering techniques. A proposed project at TRDDC to develop an Automation Design And Management System (ADAMS) aims to fulfill all these goals. We outline the architecture of ADAMS. Then we examine characteristics of PLC programs and propose a paradigm shift to view them as reactive, real-time logic programs. We suggest using logic-based techniques to write formal specifications of PLC programs. We demonstrate how these specifications can be automatically verified in terms of similarly stated requirements. We show how these specifications can be executed for off-line simulation along with powerful input data-definition facilities to understand, analyze and debug the behavior of the system. Code for any particular PLC can be automatically generated from the specifications. We illustrate these concepts by specifying a conveyor-belt system in Prolog and First Order Linear Temporal Logic (FOLTL). We discuss the pros and cons of both approaches and outline future research.

1 INTRODUCTION

Most modern factories contain automated subsystems which are discrete, modular, decentralized and hierarchical [1],[2]; we call such automation systems as Distributed Automation Systems (DiAS). Programmable Logic Controllers (PLCs)[3], with attached equipment and interconnection networks are backbones of such DiAS. We observe that a DiAS goes through a well-defined sequence of phases - Planning, Design, Implementation, Operations and Maintenance, which we call the Automation Development Life Cycle (ADLC). The tasks undertaken at each of these phases are of widely different nature. There is a clear need for an integrated software toolset to provide a comprehensive range of facilities to manage the complete life-cycle of a DiAS. We propose the name ADAMS (Automation Design And Management System) for such an integrated workbench, which is similar in spirit to a CASE toolset for software development.

In this paper, we view PLC programs as reactive, real-time logic programs and discuss the use of mathematical logic to model and specify the application and automatic verification of the properties of the given specifications. The rest of the paper is organized as follows: section 2 surveys the related work, section 3 briefly describes the motivations behind ADAMS, section 4 outlines the architecture of the ADAMS workbench, section 5 describes why and how PLC programs can be viewed as reactive, real-time logic programs, section 6 describes an example of a conveyor belt system commonly found in factories, sections 7 and 8 derive the formal specification of this system in Prolog and in temporal logic resp., section 9 compares the two specification techniques, section 10 draws the conclusions and indicates future work.

2 RELATED WORK

Formal methods are being increasingly used by software industry to state application requirements and
3 MOTIVATION FOR ADAMS

DiAS Life Cycle Management: A DiAS needs efficient resource utilization and quick adjustments for goal changes. A DiAS is a strategically important, major investment in terms of money, man-power and capital. Comprehensive and integrated management of a DiAS is critical. ADAMS will help managers, with extensive tools for exploratory planning, design and specification, to ensure that proposed DiAS is feasible, cost-effective, and meets overall planning goals. ADAMS will similarly help in managing maintenance and upgrade of a DiAS.

Relevance To India: Large-scale automation, driven by liberalization, is new to most Indian automation managers. Knowledge-based tools in ADAMS will help to plan, manage and commission large DiAS. ADAMS is specifically important and relevant to India.

Simulate Before You Implement: ADAMS allows comprehensive experimentation and simulation of proposed DiAS to maximize objectives and gain confidence. ADAMS will reduce the implementation turnaround time. Early detection of shortcomings will save trouble and money.

Uniform, Logical View Of Automation: Although the means of automation are diverse (being industry and task dependent), ADAMS provides integrated, uniform and logical views of the automation process to help in planning, analysis, design and evaluation of the proposed DiAS.

Modern Software Engineering Techniques: Common PLC programming techniques like Ladder Logic Diagram (LLD), assembly or FORTRAN-like languages lack formal foundations. Resulting programs are difficult to understand, analyze, prove properties, verify requirements, port, logically model and construct in top-down/modular way etc. Even graphical notation IEC 848 GrafCet based on Petri Nets does not solve these problems. Project management and other CASE tools (e.g. version control, data dictionary, library manager, debuggers, automatic documentation, reverse engineering tools etc.) are also not available. Advanced software engineering tools to address these problems will dramatically improve quality and reduce (unnecessarily high) costs of development and maintenance of PLC programs to control expensive and difficult processes and systems.

4 ADAMS- THE CONCEPT AND ARCHITECTURE

ADAMS is a new concept in integrated management of the life-cycle of a DiAS in a great variety of industries and services. ADAMS is envisaged to be a sophisticated, high performance, well-integrated and modular set of software tools to assist in every phase of the life-cycle of a DiAS. ADAMS provides flexible and powerful facilities to model and manage any existing or planned DiAS. ADAMS is an innovative product-line, which integrates a diverse range of software tools based on a variety of advanced and proven technologies. ADAMS clearly intends to push the state-of-the-art in automation management. Important aspects of ADAMS are:

- Knowledge-based Planner for high-level goals
- Comprehensive database of industry-standard PLCs and their associated information
- Knowledge-based Designer for DiAS design (as hierarchies and interconnections of clusters, segments and networks), layout, configuration and evaluation with respect to Planner goals
- Modular and formal specification of automation software and its verification and analysis
- Automatic generation of PLC code from specifications
- Logical simulation language to define system-level events, alarms, patterns of communication
- Distributed simulation to activate and animate entire, fully-running DiAS
- Sophisticated debugging with logical breakpoints, zoom-in, snapshots, playbacks, error-logs etc.
- Reverse engineering of specifications from PLC programs and automatic porting
Advanced software engineering tools: repository, version control, library manager, graphical simulation output and other performance analysis tools etc.

Automatic documentation of the organization of a DiAS and logic of each of its sub-systems in terms of reports, charts and diagrams.

On-line DiAS specific help for training, operation, system-understanding etc.

Intuitive graphical user interface, printer support, extensive hypertext-like intelligent on-line help

Project management tools

ADAMS will help to streamline all the activities related to the ADLC of a DiAS. ADAMS will ease DiAS management, facilitate experimentation with cost-effective DiAS designs, improve the quality and reliability of the DiAS software implementation, reduce the turn-around time in delivering a DiAS and help in maintenance and training. ADAMS will help to achieve dramatic savings in automation costs and commissioning time. ADAMS will be useful to Automation Managers (plan, control and manage life-cycle of a DiAS), System Designers (DiAS design to meet planning goals and real-life constraints), Industrial Engineers (development, implementation, testing, maintenance of DiAS software), Automation Consultants (guidance and problem-solving at all stages in DiAS life-cycle), Field Support Executives (simulate, rectify, prevent field troubles; user training), Plant Managers (training operators, engineers; efficient daily operations) etc.

5 PLC PROGRAMS AS REACTIVE REAL-TIME LOGIC PROGRAMS

A PLC program is a physical interconnection of concurrently operating hardware devices which can also be viewed as logical composition logical objects. It is an event-driven system continuously interacting with the environment. It receives input signals and generates output signals. At a logical level the various signals and their processing are in perfect synchrony. Since these are the characteristics of reactive systems [4], PLC programs are reactive systems. PLC programs also contain timing instructions and constraints. PLC programs are implicitly based on real, physical time i.e. PLC programs are also real-time systems. Hence, we feel that PLC programs can be viewed as reactive, real-time systems.

A PLC programs as LLD specifies physical connections between devices. Treating input and output values as logical variables, and devices as objects, we see that LLD specifies a logical composition of devices and also the logical behavior of each device on the basis of its inputs. The PLC has a fixed top-down, left-to-right control strategy to execute the LLD.

Logic Programming (LP) languages like Prolog [13] use predicate logic as a programming language. Some features of Prolog are: declarativeness (stressing what than how), procedural interpretation for top-down refinement, firm mathematical foundation, uniform and standard query and rule language, portability, ease of use, efficiency and powerful compilers, debuggers and Expert System tools. Prolog programs deal with predicates which are logical objects and Prolog programs are logical composition of predicates. Prolog has similar top-down, left-to-right fixed execution control strategy. Prolog has been used for describing behavior of digital circuits. Hence, Prolog-based environments seem to be attractive for specification of PLC programs. We feel that PLC programs can be viewed as reactive, real-time logic programs to make available a firm formal foundation to understand, specify and analyze PLC programs. Our view is a new approach to PLC programming and this work is based on that premise.

6 A CONVEYOR BELT SYSTEM

Conveyor belts are an important component of manufacturing systems. Figure 1 shows a simple conveyor system [14] for which we will derive the formal specifications. It is easy to construct a state-transition diagram for this system from its description given below. The conveyor separates good parts from bad parts. Each part, when moving along the conveyor, turns on a series of 5 limit switches. A good part has height 1.0" ± 0.1". LS1 signals the arrival of a part on the conveyor. A good part turns on LS2 but not LS3. When it reaches LS4, it turns on solenoid SOL1 which moves the swingarm actuator and directs the good part into another conveyor. A bad part turns on both LS2 and LS3 (too large) or fails to turn on either switch (too small). After reaching LS5, it turns on SOL2 which moves the swingarm actuator and directs the bad part into the bad parts bin.

Each time a bad part enters the bad part bin, a counter increments. When the bin is full (count complete), SOL3 turns on for a specific time (as given by timer T1) which opens the bottom of the bin and empties it. The counter resets automatically. Push-button switches START and STOP start or stop the conveyor motor starter. Watchdog timers T0 and T2 monitor the flow of parts. If a part jams, causing a delay between LS1 and LS4, then T0 times out which turns off the conveyor motor. T2 detects jam between LS4 and LS5. A conveyor indicator lamp RUN and a parts indicator lamp JAM allow remote observation.
of the conveyor's operational status. For simplification, we assume that at one time only one part is present on the conveyor.

7 SPECIFICATIONS IN PROLOG

Prolog specification of the conveyor system (appendix A) uses operational viewpoint (natural for engineers). Library IODEVICE allows the user to specify details of hardware devices in his system (e.g. limit-switches, lamps, timers, counters etc.) as Prolog facts and provides predicates for their manipulations (e.g. \texttt{chk_on, chk_off, on, off, set_timer_on, chk_timer_done} etc.). Library INSIMUL allows the user to specify the nature of the input data (e.g. distribution of good, bad and jammed parts) and automatically generates the input data to drive the simulation. Thus, the specifications can be executed (simulation of conveyor operations) to test or prove properties.

Top-level predicate \texttt{go} states that after \texttt{start_conveyor} succeeds (the conveyor is started), the conveyor continuously performs operation \texttt{do_conveyor} till \texttt{chk_stop_conveyor} succeeds (it needs to stop). Predicate \texttt{start_conveyor} is refined to state that some initial conditions must be fulfilled (\texttt{chk_startup} succeeds) before the actions to actually start the conveyor (\texttt{do_startup}) are done. Similarly, the predicate \texttt{do_conveyor} is refined to say that if the conditions for running the conveyor (\texttt{chk_running}) are OK and if a new part has arrived (\texttt{chk_part_arrival}) then process the part (\texttt{process_part}) and be ready for the next cycle (\texttt{set_next_cycle}). Refinement of \texttt{chk_stop_conveyor} says that if more data for simulation is available (\texttt{chk_more_data}) then fail else stop simulation. In the next refinement, predicates \texttt{chk_startup, do_startup} specify the actual startup conditions as well as actions to start the conveyor. Similarly, all other predicates in the previous level are refined. This clean process of top-down refinement of specifications continues until all aspects of the system's behavior are specified. This Prolog program can now be executed and its properties verified automatically.

8 SPECIFICATIONS IN TEMPORAL LOGIC

\textit{Temporal Logic} is a formal system based on mathematical logic for qualitative representation and reasoning about time and temporal behavior of systems. \textit{First-order Linear Temporal Logic (FOLTL)} \cite{5} is a one-sorted, point-oriented first-order logic. FOLTL supports a linear model of time where every moment has a unique successor. It also presumes the time to be discrete. Although it is not oriented towards real-time systems, we have chosen it for its simplicity. FOLTL adds temporal operators in Table 1 to first-order predicate logic to represent and reason about how truth of an assertion changes with time. We assume that all function, constant and predicate symbols are nontemporal and propositions and variables may be temporal or nontemporal. \cite{5} defines the exact syntax and semantics of FOLTL. Temporal logics have been used to specify and verify nonterminating concurrent and reactive programs like protocols, Operating Systems etc.

Appendix B gives the specification of the conveyor system in FOLTL. These specification are written from the point of view of identifying events and their processing; e.g. the specification for conveyor startup says that if at any time the \texttt{START} button is pushed and \texttt{STOP} button is not pushed then there comes a time when motor is started and there comes another moment when the run lamp is put on. All other specifications can be similarly and easily understood.

9 COMPARING PROLOG AND FOLTL

Prolog allows a clean process of top-down refinement of logical specifications which can be executed and its properties verified automatically. The specifications start at logical level and are refined successively till they reach the hardware level. This logical view of the system is absent in LLD and other approaches for PLC programming. Prolog also offers advantages like popular, portable, efficient development environments, clean and simple semantics etc. Prolog also allows clean and powerful off-line simulation, debugging and analysis facilities. Simple properties of the system which always hold (e.g. good swingarm and bad swingarm never open simultaneously) or which hold at specific time-points (e.g. JAM light comes on and conveyor stops after a jam occurs) can be easily verified. However, it is difficult to use Prolog to verify more complex properties which hold over conditionally defined intervals. FOLTL is clearly powerful and expressive. However, it lacks some features necessary for real-time systems as well as it does not have efficient techniques for execution of specifications and for verification of its properties. Stepwise development of operational specifications is difficult in such logics.

10 CONCLUSIONS AND FUTURE WORK

We feel that reactive, real-time extensions to Prolog will effectively combine the advantages of Prolog and temporal logic. We also feel that such a language should have powerful abstraction and object-oriented features to enable building of complex and reusable specifications. We are
currently defining such a language, its temporal features and its semantics. Development of ADAMS will shortly be undertaken.

ACKNOWLEDGMENTS

We are grateful to Dr. E.C. Subbarao for his constant encouragement. We acknowledge the interest and help from A.K. Gupta and members of his Process Control Group - K. Krishna, Milind Joshi, Sanjeev Dange, Nivedita Mitra and others. Hanumantha Rao, Pavan Kumar, Vivek Balaraman, Shailesh Abhyankar, Dr. Balachandran, Vinay Kulkarni and many others at TRDDC have been enthusiastic and freely contributed their time and ideas.

REFERENCES


Table 1. Some common linear temporal operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sometime/eventually</td>
<td>Fp is true now iff at some future moment p is true</td>
</tr>
<tr>
<td>always/henceforth</td>
<td>Gp is true now iff at all future moments p is true</td>
</tr>
<tr>
<td>nexttime</td>
<td>Xp is true now iff p is true at the next moment</td>
</tr>
<tr>
<td>until</td>
<td>(p U q) true now iff q does eventually hold and p will hold everywhere prior to q</td>
</tr>
<tr>
<td>precedes/before</td>
<td>(p B q) true now iff q occurs later then it is strictly preceded by occurrence of p</td>
</tr>
</tbody>
</table>
APPENDIX A. PROLOG SPECIFICATIONS OF THE CONVEYOR SYSTEM

Note: Some low-level predicates are not shown.

```prolog
\( \text{go} :- \text{start_conveyor}, \)  
  repeat, \( \text{do_conveyor}, /* process one part */ \)
  \( \text{chk_stop_conveyor}, /* process next part if no need to stop */ \)
\( \text{start_conveyor} :- \)
  \( \text{chk_startup}, /* \text{chk} that conveyor is ready to be started */ \)
  \( \text{do_startup}. /* \text{start the conveyor */} \)
\( \text{do_conveyor} :- \)
  \( \text{chk_running}, /* \text{chk} that conditions to run conveyor are OK */ \)
  \( \text{chk_part_arrival}, /* \text{chk} that the part arrives on the conveyor */ \)
  \( \text{process_part, /* process the arrived part */} \)
  \( \text{set_next_cycle}. /* \text{be ready for the next cycle */} \)
\( \text{chk_stop_conveyor} :- \)
  \( \text{chk_more_data, /* \text{input} data for next cycle is present */} \)
  \(! \), fail. /* do not stop as conveyor's next cycle can be run */
\( \text{chk_stop_conveyor} :- \text{write('No more input data! Conveyor simulation over.'), nl.} \)
\( \text{chk_startup} :- /* \text{TRUE if conveyor is ready to be started */} \)
  \( \text{chk_on( read, start ), /* \text{START} pushbutton must be pushed */} \)
  \( \text{chk_on( read, stop ). /* \text{STOP} pushbutton must not be pushed */} \)
\( \text{do_startup} :- /* \text{start the conveyor */} \)
  \( \text{on( motor_starter ), /* \text{start} conveyor motor */} \)
  \( \text{on( motor_starter_aux ), /* \text{start} motor starter aux */} \)
  \( \text{on( run )}, /* \text{switch} on the RUN indicator lamp */ \)
\( \text{chk_running} :- \text{chk_start, /* \text{START} is pushed or motor_starter_aux is ON */} \)
  \( \text{chk_on( read, stop ), /* \text{STOP} push-button must not be pushed */} \)
  \( \text{chk_off( read, jam )}, /* \text{chk} that JAM indicator is OFF */ \)
\( \text{chk_running} :- \text{write('Conveyor stopped! Re-start data not available!'), nl, abort(0).} \)
\( \text{chk_part_arrival} :- \text{chk_on( read, Is1 ), /* limit-switch 1 must be ON */} \)
  \( \text{set_timer_on( t0 ), /* \text{start} watchdog timer t0 */} \)
  \( \text{set_timer_on( t2 ), /* \text{start} watchdog timer t2 */} \)
\( \text{process_part} :- \text{separate_good_part}. /* \text{separate} part as per whether it is good or bad */ \)
\( \text{process_part} :- \text{separate_bad_part}. \)
\( \text{chk_start} :- \text{chk_on( read, start ). /* \text{START} pushed or motor_starter_aux ON */} \)
\( \text{chk_start} :- \text{chk_on( get, motor_starter_aux )}. \)
\( \text{separate_good_part} :- \text{chk_good_part, process_good_part}. \)
\( \text{separate_bad_part} :- \text{chk_bad_part, process_bad_part}. \)
```

Figure 1. Conveyor Belt: separating good parts from bad parts.
chn_good_part :- above_min_height, below_max_height.
process_good_part :- chn_jam_good_part, remove_jam_good_part,
write("Good part JAMmed! Stopping the conveyor!!"), nl.
chn_bad_part :- not(chn_good_part).
process_bad_part :- chn_jam_bad_part, remove_jam_bad_part, write("Bad part JAMmed! Stopping the conveyor!!"), nl.
chn_bad_part :- remove_bad_part, write("Bad part found!"), nl.
chn_bad_part :- open_bad_bin_bottom :- write("Opening the bottom of the bad part bin!"), nl, on(sol3).
chn_bad_part :- dump_bad_bin.
chn_bad_part :- actuate_good.swingarm.
chn_bad_part :- off(sol5), off(motor_starter), off(jam).
chn_good_part :- on(sol1), off(sol2), on(sol4).
chn_good_part :- remove_part.
chn_good_part :- actuate_bad.swingarm, inc_bad_counter, dump_bad_bin.
chn_good_part :- remove_part.
chn_good_part :- write("Opening the bottom of the good part bin!").
actuate_good.swingarm :- chn_on(read, ls4), on(sol1), off(sol4).
actuate_good.swingarm :- chn_on(read, ls5), off(sol5).
dump_bad_bin :- chn_counter(bad_ctr, done), open_bad_bin_bottom.
dump_bad_bin.
open_bad_bin_bottom :- write("Opening the bottom of the bad part bin!").
chn_good_part :- off(sol3).

APPENDIX B. FOLTL SPECIFICATIONS OF THE CONVEYOR SYSTEM

conveyor start: (chn_off(start) ∧ X(chn_on(start) ∧ chn_off(stop))) ⇒
Fchn_on(motor_starter) ∧ Fchn_on(motor_starter_aux) ∧ Fchn_on(run)

conveyor stop: (chn_on(stop) ∨ chn_on(jam)) ⇒ Fchn_off(motor_starter) ∧ Fchn_off(motor_starter_aux) ∧ Fchn_off(run)

part arrival: on(ls1) ⇒ Fchn_timer_on(t0) ∧ Fchn_timer_on(t2)

good part movement: chn_timer_on(t0) U chn_on(ls4) ∨ chn_timer_timeout(t0)

good part jam: chn_timer_timeout(t0) ⇒ Fchn_off(motor_starter) ∧ Fchn_off(motor_starter_aux) ∧ Fchn_off(run) ∧
Fchn_on(jam)

good part processing: (chn_on(ls2) ∧ chn_off(ls3) ∧ chn_on(ls4)) ⇒ Fchn_on(sol1) ∧
Fchn_off(ls2) ∧ Fchn_off(ls4) ∧ chn_timer_off(t0)

bad part movement: chn_timer_on(t2) U (chn_on(ls5) ∨ chn_timer_timeout(t2))

bad part jam: chn_timer_timeout(t2) ⇒ Fchn_off(motor_starter) ∧ Fchn_off(motor_starter_aux) ∧ Fchn_off(run) ∧
Fchn_on(jam)

bad part processing: (¬chn_on(sol2) ∧ chn_off(sol3) ∧ chn_on(sol5)) ⇒ Fchn_on(sol2) ∧
Fchn_off(sol2) ∧ chn_off(sol3) ∧ chn_off(sol5) ∧ chn_timer_off(t0)

(chn_on(sol2) ∧ chn_off(bad_ctr) ⇒ Fchn_counter_on(bad_ctr) ∧ Finc_counter(bad_counter)
chn_on(sol2) ∧ chn_on(bad_ctr) ⇒ Finc_counter(bad_counter)
chn_counter_done(bad_ctr) ⇒ Fchn_timer_on(t1) ∧ Fchn_on(sol3) ∧ Fchn_timer_off(t1)
∧ Fchn_off(sol3) ∧ Fchn_counter_off(bad_ctr)