Int J Adv Manuf Technol (2005) 26: 436–447
DOI 10.1007/s00170-003-1996-0

ORIGINAL ARTICLE

M.R. Lucas · D.M. Tilbury

Methods of measuring the size and complexity of PLC programs in different logic control design methodologies*

Received: 26 June 2003 / Accepted: 8 October 2003 / Published online: 3 August 2005

Abstract Currently there is a wide variety of logic control design methodologies used in industrial logic design. These methodologies include ladder diagrams, function block diagrams, sequential function charts, and flow charts, but driven by a desire for verifiability, academics are developing additional logic control design methodologies, such as modular finite state machines and Petri nets. Using these, important properties of programs can be verified and some logic can be generated automatically from a part plan. The main contribution of this paper is to define methods for measuring programs written in different methodologies, so that the performance of the methodologies can be compared.

We demonstrate these methods of measurement using four program samples that perform similar functions on the same machine, written in four logic control design methodologies: ladder diagrams, Petri nets, signal interpreted Petri nets and modular finite state machines.

Keywords Comparison of logic design methods · Logic control · PLC program complexity

1 Introduction

In the automotive manufacturing industry, machines are becoming increasingly complex. This complexity is driving up the cost of the controls required to ensure each machine is as safe and productive as possible.

Standard methods of writing control logic are being pressed to their limits, and a number of academic alternatives have been proposed, which may be more efficient and more reliable. However, there is currently no consensus on the merits of any particular logic control design methodology, and no method of measuring or comparing them.

There are many ways that the effectiveness of a logic control design methodology could be measured. Some measures include: the number of elements required to create a certain program, the ease of extracting information from an existing program, the time required to create a program, the amount of reuse typical in a certain methodology, the time and manpower required to install and debug a program on a machine, and the time and manpower required to change an existing program.

Many methods of measuring the effectiveness of a logic control design methodology require that an industrial-sized project be developed, installed, debugged, and modified. This in turn requires the prior development of a fully featured development environment. In short, these methods of measurement could be prohibitively expensive.

In contrast to this full development, we have chosen to measure two properties: the number of elements required to generate a particular program, and the difficulty of answering certain questions based on an existing program. These can be used to measure existing programs using uniform metrics. However, high quality development environments are not required. For example, the modular finite state machine sample discussed in Sect. 4.5 was primarily developed using pencil and paper.

In the next section we will review previous efforts to measure logic control design methodologies, and summarize the methodologies that we will use as demonstrations. In Sect. 3 we will present the measurements developed. In Sect. 4 we will demonstrate the measurements on previously published code samples written using four logic control design methodologies. Then Sect. 5 discusses the measurements and possible future work.

2 Review of relevant literature

There are two main areas of research that are relevant to this work. First we will review several logic control design methodologies that will be used as examples in Sect. 4, and then we will

M.R. Lucas (✉) · D.M. Tilbury
Engineering Research Center for Reconfigurable Manufacturing Systems,
Department of Mechanical Engineering,
University of Michigan,
Ann Arbor, MI 48109-2125, USA
E-mail: mrlucas@umich.edu
*This research was supported in part by the NSF under grant EEC95-29125.

review existing methods of measuring the effectiveness of logic control design methodologies.

2.1 Summary of logic control design methodologies used

As mentioned previously, there are many logic control design methodologies available for logic control development. The IEC 61131-3 standard includes five languages. Ladder diagrams are the most common, and will be described below. Sequential functions charts are a method of controlling the execution of program segments, and are used in some specific problems. Function block diagrams are a method of programming using data flow graphs; although they are rarely used now, they are the basis for the emerging standard 61499 [1]. In addition instruction list and structured text are text based languages, roughly analogous to assembly and Pascal, respectively. They are rarely used in the U.S. In addition, a nonstandard flow chart language is used by some developers [2].

In academia, alternative logic control design methodologies have been developed. Most of these are based on Petri nets, a well-known method of analyzing manufacturing systems, which can be adapted for control. Petri net methods should be easy to write and debug while allowing some structure. In addition some work has been done using modular finite state machines. Modular finite state machines should allow for substantial structure and code reuse.

For this paper we will study programs written using four logic control design methodologies: ladder diagrams, Petri nets, signal interpreted Petri nets, and modular finite state machines. These methods were chosen to compare a reasonable breadth of academic methodologies as well as the industry standard ladder diagrams. In this section we provide details on each of the logic control design methodologies considered in this paper.

2.1.1 Ladder diagrams

Ladder diagrams are the primary industrial logic control design methodology used in American industry today [3]. This method is the end result of a gradual evolution from the physical relays, which electricians had previously used to control machining systems. A ladder diagram consists of individual rungs, which are executed sequentially (see Fig. 1). In general, each rung is the sole control for a single output or internal state variable. Internal state variables are minimized to preserve as simple a path as possible between inputs and outputs.

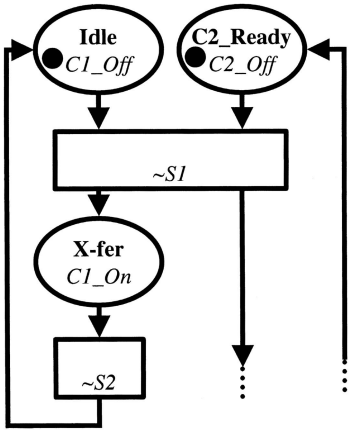

Fig. 2. Example of a portion of a Petri net. The ovals are called *places*, and each can hold one or more *tokens*, represented by the small dark circles. The places and *transitions* are connected by directed arcs under the restriction that each arc connects a place and a transition. When a transition fires, it removes one token from each place with an arc to the transition, and adds one token to each place with an arc from the transition. A transition will fire whenever its condition (usually a sensor value, in italics) is true, and firing will not cause any places to have a negative number of tokens. Output (in italics) is generated by the places whenever they contain at least one token

2.1.2 Petri nets

Petri nets are well established in academia as a means of modelling discrete event systems. They are particularly useful for systems that exhibit parallel and concurrent operations [4].

Petri nets can be extended to provide for active control of systems by assigning inputs and outputs to the places and transitions of the net (see Fig. 2). A program that implements this concept has been written at the University of Kentucky [5, 6] and methods of generating and verifying Petri net controllers have been developed by E. Park et al. [7, 8].

The program studied in this paper was written using the tools developed in [5, 6] and using the format developed by E. Park. The complete program is contained in [9].

2.1.3 Signal interpreted Petri nets

Signal interpreted Petri nets (SIPNs) are a variation on the standard Petri nets framework developed by Frey et al. [10, 11] (see Fig. 3).

The primary differences between SIPNs and standard Petri nets are:

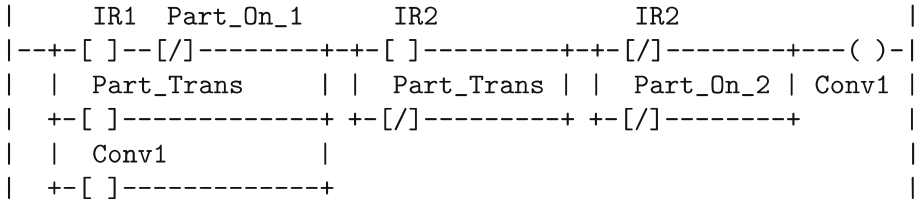

Fig. 1. Example of a single rung of a ladder diagram. This is the rung used to control the `Conv1` output. `Conv1` is turned on by either `IR1` and `NOT Part_On_1`, or by `Part_Trans`. It is turned off by either `NOT IR2` and `Part_Trans` or `IR2` and `Part_On_2`. This rung comes from an unpublished program that uses two conveyors to transport parts, under the constraint that no conveyor ever contains two parts simultaneously.

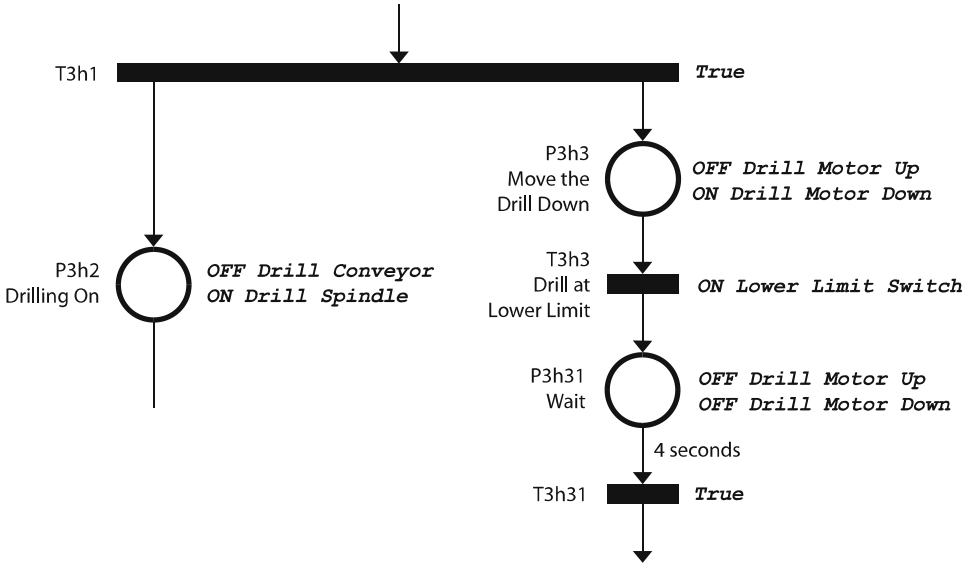

Fig. 3. Example of a portion of a signal-interpreted Petri net. Transitions fire when their conditions are true, and firing will not cause any transition to have less then zero or more then one token. Outputs are generated by combining the output conditions of all active places. Hierarchical nets are allowed (not shown)

Evolution A transition in SIPN will only fire if there is one token in each in-place, and there are no tokens in any out-place. Therefore no place will ever contain multiple tokens. In addition, if the firing of one transition enables another to fire, the second will fire during the same scan cycle. The absence of racing conditions resulting from this firing rule can be verified by the development environment.

I/O A Boolean equation on input signals may be placed on a transition as a firing condition. Each place defines the state for each system output as either 0 (off), 1 (on), or − (don't care). The actual output is the sum of the outputs of each place that contains a token. The development environment ensures that the output is fully defined and not contradictory.

Hierarchy The SIPNs used to generate this program allow hierarchy. A subnet may be placed within a single place of a Petri net. Conditions are defined in [10] to ensure deterministic behavior.

The program studied in this paper was developed by Klein [12]. The complete program is shown in [13].

Additional variants on Petri nets are occasionally used in literature. For example, Uzam et al. [14] use Petri nets with inhibitor arcs to control a model system. They use reachability graphs to validate the system, and then generate ladder diagrams via "token-passing logic." Peng and Zhou [15] survey the state of research regarding conversion between Petri nets and ladder diagrams, and generally find conversion schemes lacking.

2.1.4 Modular finite state machines

Modular finite state machines are an extension of the standard finite state machine formulation. A modular finite state machine program consists of a set of modules, each of which contains a trigger/response finite state machine, and instructions for communicating with other modules. This method attempts to preserve the formality and verifiability for finite state machines in a modular framework (see Fig. 4). Details can be found in [16].

The program studied in this paper was created using the software tools developed in [17]. A report on this coding effort is in [18].

2.2 Existing logic measurement methods

A few attempts at a more direct comparison of logic control design methodologies have been made, although these comparisons are all in a restricted domain.

Venkatesh et al. [19] devised a method of comparing the complexity of programs written using ladder diagrams and Petri nets using a "basic element approach." Their method was based on counting the number of elements required to represent a particular program. An "element" was chosen to be a place, transition, or arc for Petri nets or a contact, coil, rung or branch for ladder diagrams.

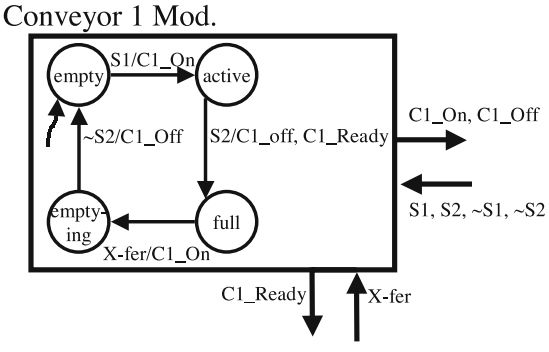

Fig. 4. Example of a single module of a modular finite state machine. This module has four states, exactly one of which is always active. Trigger events arrive from one of the two ports, and cause a transition to fire. Firing a transition can cause a change in the active state and/or the transmission of a response event. Ports can connect either to the physical I/O or to another module. For example, if the module is in the "empty" state, and event S1 occurs, the module transitions to the "active" state and the event C1_On will be sent out the right-hand port (either to the environment or to another module)

A somewhat more sophisticated method of measuring the complexity was presented by Lee and Hsu [20], which converts the Petri and ladder programs into Boolean expressions, and then counts the number of Boolean operators and equations required.

Both of these methods found that Petri nets were more efficient than ladder diagrams for the samples tested.

In addition to methods of comparing logic control design methodologies, there are a number of ways of measuring traditional, text-based languages. The most common metric is the lines of code (LOC) required to create a program. A number of more complex measurement methods have risen based on "software science" [21], which counted the number of operators and operands required, and how often they are used. Conte et al. [22] discusses many of these methods of measurement, which typically involve measuring code based on either the number of lines, operators, operands, functions, modules or similar objects. These measurements can then be used to estimate size, number of errors, or time required, usually using empirically derived equations. For example, one such derived equation is $S_s = 102 + 5.31\,\text{VARS}$, where S_s is the size of the program in lines of code, and VARS is the number of variables required. These methods have provided insights into how programs are written. However, they are of limited use when evaluating a new programming language since empirical data can vary. In addition, it is not clear how they apply to the more graphically based logic control design methodologies.

The metrics presented in this paper can be used to measure the logic in a manner that is consistent across logic control design methodologies.

3 Methods of analysis

To analyze the generated programs we will use two methods: direct measurement and examination of which data can be easily retrieved as described below.

3.1 Direct measurement of programs

In traditional programming languages (such as C, C++ or Pascal) the complexity of a piece of code is generally measured by the number of lines. However, since these logic control design methodologies are quite different from one another, common elements must be found to allow for reasonable measurements. The common elements that we define are: operation, state variable, cause of operation, effect of operation, and module.

Definition 1 (Operation). *A single, inseparable action or set of actions that can be performed by the program.*

Definition 2 (State variable). *A single object that maintains state.*

Definition 3 (Cause of operation). *If an operation X can enable or disable an operation Y, then X is a cause of Y.*

Definition 4 (Effect of operation). *If an operation X can enable or disable an operation Y, then Y is an effect of X.*

Definition 5 (Module). *A set of operations that are grouped by the program designer to perform a function is called a module. Note that in some logic control design methodologies modules are very well defined, and in others they are only defined by their positioning and comments.*

Interpretations of these terms for each methodology considered in this paper are shown in Table 1. These terms are used to define the following three measures of complexity for a single piece of code:[1]

[1] The meanings for these three measures were developed in [23] as part of a framework for subjectively evaluating visual programming environments, although formal metrics were not defined. In that paper they were referred to as diffuseness, abstraction gradient, and hidden dependencies. The names have been changed to size, modularity, and interconnectedness to make the concepts easier to understand.

Table 1. Interpretations of terms for different programming languages. Each of the logic control design methodologies can be broken into similar parts as shown in this table. These definitions (operation, state variable, cause of operation, and effect of operation) will be used extensively when discussing measurement details

	Ladder diagram	Petri net	Signal interpreted Petri net	Modular finite state machine
Operation	A single grouping of sets/resets and the logic which controls their implementation	A single transition and the annotation associated with the transition and destination states	A single transition and the annotation associated with the transition and attached states	A single transition as well as the annotation associated with that transition
State variable	Any internal or output bit	A single place	A single place	A single state
Cause of operation X	Any input or operation which sets a bit which is a condition on X	Any input which is a condition on X, or any operation with an out-place that is an in-place of X	Any input which is a condition on X, or any operation attached to a place to which X is attached	Any input which is a trigger on operation X, or any operation whose response is a trigger on X
Effect of operation X	Any output which is set by X or operation whose conditions contain a bit set by X	Any output which is set by an out-place of X, or any operation whose in-places contain an out-place of X	Any output which is set by an out-place of X, or any operation attached to a place to which X is attached	Any output which is a response to X, or any operation whose trigger is a response to X
Module	A set of related rungs grouped by their position and/or comments	A set of related places and transitions grouped by their positions and/or comments	A single net or subnet	A single module

Measurement 1 (Size) The size of a piece of code can be measured two ways: the number of operations in the code

$$N_o = \text{number of operations}, \tag{1}$$

and the number of state variables in addition to the I/O definitions

$$N_s = \text{number of state variables}. \tag{2}$$

The following definition is not a measure, but will be used in further definitions.

Definition 6 (The size of a module). *A module is a conceptual unit of code that is generally less than the whole. The number of operations in module i will be denoted as:*

$$N_o^i = \textit{number of operations in module } i. \tag{3}$$

This will be used to determine the size of a module, since the number of state variables in a module is not always well defined.

Measurement 2 (Modularity) The modularity of a piece of code will consist of two measures: the number of modules in the piece of code

$$N_m = \textit{number of modules}, \tag{4}$$

and the size of the largest module in relation to the size of the entire code as measured by number of operations:

$$S = \frac{\max N_o^i}{N_o}. \tag{5}$$

More abstract code (with more, smaller modules) is generally easier to reconfigure and maintain, although it can sometimes be more difficult to understand.

Measurement 3 (Interconnectedness) Interconnectedness consists of two measures: the number of possible causes for an operation, averaged over the number of operations:

$$n_c^i = \textit{number of possible causes for operation } I, \tag{6}$$

$$IC_c = \frac{1}{N_o}\sum_{i=1}^{N_o} n_c^i, \tag{7}$$

and the number of possible effects for an operation, averaged over the number of operations:

$$n_e^i = \textit{number of possible effects for operation } i \tag{8}$$

$$IC_e = \frac{1}{N_o}\sum_{i=1}^{N_o} n_e^i. \tag{9}$$

As the interconnectedness of a program decreases, it is likely to be easier to understand and debug.

The direct measurements will determine the complexity of the code generated, and should correspond with development time and cost.

Table 2. Description of scale used to evaluate the accessibility of data in the various logic control design methodologies

Value	Description
Easy	No search of the entire code or mental simulations needed
Moderate	Searching through most of the code and/or simple mental simulations are needed
Hard	Either multiple searches through the entire code or complex, multi-state simulations are needed

3.2 Accessibility of data

Another measure of each logic control design methodology is the accessibility of the information in the program. That is, how well can a programmer solve problems using the code. To measure accessibility we define four general styles of questions, which a designer may attempt to answer using the logic. We also define a method of describing the difficulty of answering these questions.

To utilize this measurement method, a researcher must first create specific questions based on the particular application, then examine the methods of answering these questions using the tools that would likely be available. Examples will be given in Sect. 4.

The following types of information are typically required of a portion of logic:

Single output debugging Specific questions regarding specific unexpected behavior in the machine.

System manipulation Questions regarding how the user can manipulate the machine to achieve a desired state.

Desired system behavior Questions regarding the desired behavior of the machine when examining only the schematics and the logic.

Unexpected system behavior Questions regarding the system's response to unexpected events.

While these scenarios certainly do not represent all questions that may be asked about a program, they represent a reasonable variety of information that may be required from a program.

The difficulty of answering the question in each scenario will be judged as easy, moderate or hard according to the scale in Table 2.

4 Demonstration of measurements

To demonstrate these measurement techniques we will use four existing programs. These programs were generated at separate times by separate people, and therefore do not constitute an experimentally controlled set. However, they have all been designed to control similar actions on the same test-bed. In this section we will analyze each program individually according to the methods described in Sect. 3.

Each program was developed to control the flexible manufacturing test-bed (see Fig. 5). This test-bed has a drilling station, a vertical milling station with a tool changer, and a horizontal

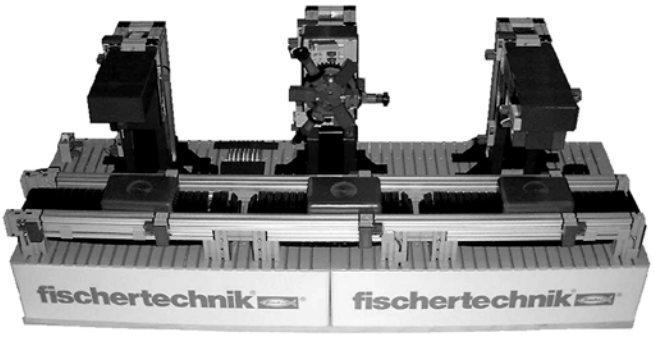

Fig. 5. Flexible manufacturing test-bed

milling station. The parts can be moved from station to station by three conveyor belts. This system has 15 binary outputs and 15 binary inputs. For more information on this test-bed see [24].

The desired part plan is, for each part: drill once, operate with each tool in the vertical tool changer, and then perform two passes with the horizontal mill. Parts should be moved to the next station as soon as it is clear.

The programs were generated prior to this study and most have been previously published.

4.1 Specification of accessibility scenarios

The accessibility of the program data will be determined by analyzing the difficulty of answering the following questions about the controlled system. These scenarios are specific examples of types of scenarios described in Sect. 3.2.

Scenario 1 (Single output debugging)
Situation: The system is currently running, and the first conveyor has not turned on when a new part was placed at the start.
Question: Why hasn't the drill conveyor turned on?

This sort of question is often used as an example of why ladder diagrams are so simple.

Scenario 2 (System manipulation)
Situation: The system was processing a single part when that part was removed unexpectedly. The user must now manually depress sensors as needed to manipulate the system state.
Question: What needs to be done to return the system to the "idle" state?

This should not be tried with a real system due to safety considerations, but it is a reasonable approximation of situations that occur in industrial systems.

Scenario 3 (Desired system behavior)
Situation: The user only has access to the logic and a description of the machine.
Question: What happens to a part after it has been drilled?

Scenario 4 (Unexpected system behavior)
Situation: The user only has access to the logic and a description of the machine.
Question: What happens if an additional part is added to the vertical mill conveyor mid-stream?

4.2 Analysis of a ladder diagram solution

The ladder diagram was professionally generated by the manufacturer of the test-bed [25]. The complete code can be found at [24].

4.2.1 Direct measurements

The ladder diagram contains 27 separate rungs of code, and each rung represents a distinct operation; therefore $N_o = 27$. The ladder diagram requires four latches (Boolean state variables), two counters and two timers. In addition, each of the 15 outputs can be read by any rung, and effectively becomes a state variable. Therefore there are 23 state variables, therefore $N_s = 23$.

The rungs in the ladder diagram do not follow an obvious order, and there is no separation of the ladder into separate parts. Therefore there is only one module, containing 100% of the code, therefore $N_m = 1$ and $S = 1.0$.

In order to find all the causes or effects of a single rung, each rung of code must be searched in turn. Therefore for this piece of code, 27 separate operations must be searched to find all the causes or effects of a single operation. However, since each output or state variable is controlled from a single rung, once a programmer is familiar with the code he will be able to turn directly to a particular rung. Since there are an average of 4.2 elements per rung, only 4.2 rungs on average need to be searched to find all possible causes. Therefore $\mathrm{IC}_c = 4.2$ and $\mathrm{IC}_e = 27$. Note that many ladder editors provide a "cross reference table", which maintains a list of all rungs that are affected by an output or state variable, in addition to providing direct access to the rung that maintains the output or state.

These numbers are summarized in Table 3.

4.2.2 Accessibility of data

Scenario 1 (Single output debugging)
In this ladder diagram (as in most), each output is controlled by exactly one rung, the position of which is usually known. Therefore, to determine why a particular rung has not turned on, one needs to find the single appropriate rung, and examine the inputs and state variables that affect it. In fact, some ladder programming systems highlight the elements that are logically true, so

Table 3. Direct Measurement Summaries: These terms are defined in Sect. 3.1. Lower numbers represent smaller and/or simpler programs

	Size		Modularity		Interconnectness	
	N_o	N_s	N_m	S	IC_e	IC_c
Ladder diagram	27	23	1	1.00	27	4.2
Petri net	49	63	3	0.38	2.18	2.02
SIPN	50	68	7	0.20	3.30[a]	3.66
Modular FSM	128	80	19	0.16	8.73	9.60

[a] Using a slightly different definition of effect this number is 7.34. We believe that the number 3.30 most accurately represents this value. See Sect. 4.4.1 for more details

that a quick visual scan can determine the cause of the problem. In many cases the problem can be pinpointed to a single input that is in the wrong state.

Since this problem requires neither searching the code or complex simulation, we will judge it easy.

Scenario 2 (System manipulation)

While it seems very easy to determine the cause of a single unexpected output, it is not clear how the rungs relate to each other. This means that in general, it is very difficult to plan more than one step into the future without substantial understanding of the nature of the system.

However, this ladder diagram contains a very prominent latch called the `Reset Latch`, which appears in 12 rungs. It is clear from a casual reading of the code that the `Reset Latch` will cause the system to return to the "idle" state when the system is powered down or the on/off switch is turned off.

Since this problem did not require any hard operations, it may be judged easy. It should be noted that this problem was solved by the variable names chosen for the program. A similar problem (e.g., cause the system to continue as if a removed part never existed) would be require the operator to manually simulate the entire ladder diagram, a hard operation.

Scenario 3 (Desired system behavior)

It is difficult to determine what will happen to the part after it has been drilled at the first station. There is a latch called the `Drill Done Latch`, which appears in the rungs controlling the `Vertical Mill Conveyor`, the `Drill Press Conveyor Motor`, and the drill press motors (both up and down). It seems reasonable to assume that this latch is important to understanding this scenario, see Fig. 6.

This variable is set when the drill press has been on and the lower limit switch for the duration specified in the `Drill Press Timer` (requires knowledge of the rung controlling the `Drill Press Timer`, not shown). It is unset by the `Reset Latch`. It can also be unset both by having no part at the `Drill Press IR Sensor` and either a part at the `Vertical Mill Position Switch` or the vertical mill leaving it's upper position. That is, the drilling is considered "done" from when the drill operation is completed to when the part has left the drill station and arrived at the vertical mill station. Therefore it stands to reason that the part moves to the vertical mill station after it has been drilled.

This kind of reasoning must be continued through five additional rungs before the actions on a part can be fully understood. The five rungs control: `Vertical Mill Head Motor Down`, `Vertical Drill Down Latch`, `Vertical Mill Conveyor Motor`, `Vertical Mill Rotate Motor`, and the `Drill Count` counter.

This scenario required the mental simulation of multiple independent rungs of the logic. Therefore we will judge this problem hard.

Scenario 4 (Unexpected system behavior)

If an unexpected part is added where it can be detected by the `Vertical Mill Position Switch` when the system is waiting for a part to arrive there, the effects are easy to understand, since the system cannot distinguish the unexpected part from the expected one. However, this will leave another part in the system. This can be determined just from knowledge of the system, without consulting the ladder diagram.

If an unexpected part arrives when no part is expected, it could potentially affect all rungs that read that sensor. There are four such rungs. That sensor is used to:

1) Turn off the `Drill Press Conveyor Motor` when a part arrives

2) Turn off the `Vertical Mill Conveyor Motor` when a part arrives

3) Turn off the `Vertical Drill Down Latch` when a part leaves

4) Ensure that the vertical mill head only lowers when a part is present

In each of these cases, the arrival of an unexpected part does not immediately cause a change of state. In addition, various interlocks (such as condition 4 above) ensure that certain forbidden operations will never take place.

However, while it is not too difficult to determine that the arrival of an unexpected part will not cause the system to crash, or forbidden operations to occur, it is not clear from the code if the extra parts on the conveyor will ever clear themselves out, or if there will always be an extra part between the drill and the vertical mill.

To solve this without running the program, the programmer would need to simulate mentally both the entire program and the test-bed for many cycles. Therefore we will judge this problem hard.

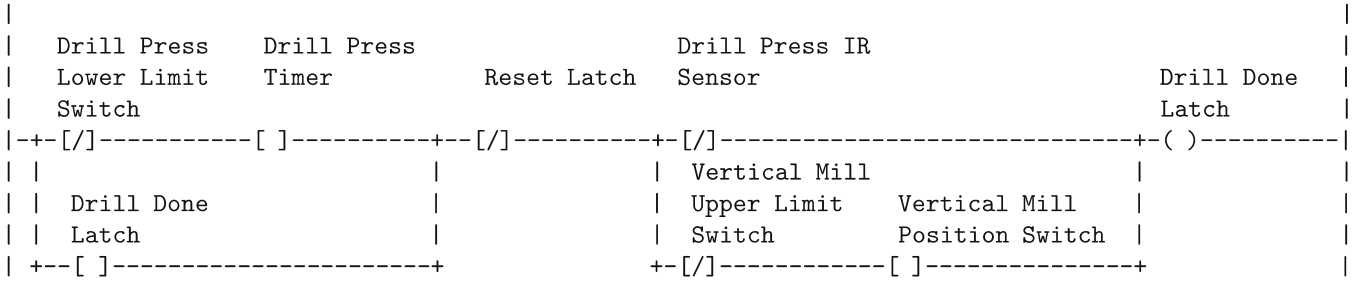

Fig. 6. Example of a single rung from the sample program. The `Drill Done Latch` will turn on when the drill press reaches its lower limit switch, and remains there for the duration of the drill press timer. It will turn off when the `Reset Latch` is set, or the part leaves the drill press sensor and either reaches the vertical mill position switch, or the vertical mill leaves its upper limit switch. (All physical switches are normally closed.)

4.3 Analysis of a Petri net solution

The Petri net program for the test-bed was generated by Gollapudi [9]. A portion of the logic is shown in Fig. 7.

4.3.1 Direct measurements

The Petri net contains 63 places and 49 transitions. Therefore the $N_o = 49$ and $N_s = 63$, since all states and operations are directly represented by places and transitions. The program consists of three modules. It has a module for the drill station, the vertical mill station, and the horizontal mill station. The largest module has 19 operations, or 0.38 of the total. Therefore $N_m = 3$ and $S = 0.38$.

The interconnectedness values are: $\mathrm{IC}_c = 2.02$ and $\mathrm{IC}_e = 2.18$. The program consists primarily of transitions with one condition between two places with one output. Therefore most of the transitions have two possible causes (the condition on the transition and the previous transition) and two possible effects (the output on the out-place and the next transition).

These values are summarized in Table 3.

4.3.2 Accessibility of data

Scenario 1 (Single output debugging)
In a Petri net a particular output can be turned on from a variety of places. To determine why the drill conveyor had not turned on in this particular case, it would be necessary to determine the place that was supposed to be active at this point by searching the code. After finding the desired place, the programmer would need to determine what condition was needed to activate that place from the current state. Finding the desired place would require a search of all 16 places within the appropriate module. Then the user would need to trace back to the program path to the current state, and determine the unsatisfied condition.

This requires a search of the entire code for the desired state, followed by relatively simple reverse search for the missing condition. This problem will be judged moderately difficult.

Scenario 2 (System manipulation)
In a Petri net the flow of the system can usually be determined easily from the layout of the program. Therefore, given a current state, and a desired state, it is generally straightforward to determine the quickest path to the desired state, and the programmer then needs to check the condition of each transition in turn.

Therefore, since this requires neither searching nor mental simulation, this problem is easy.

Scenario 3 (Desired system behavior)
Since Petri nets express a sequence of events, most sequential data is readily available. For example the question "What does the drill do after it moves down?" is directly available from the diagram. However sequential data based on the part is not as readily available, since it relies on knowledge of the physical system. In this case, after the part has been drilled the program waits for a synchronizing operation, and then turns on the drill conveyor. The part then triggers a sensor in the next module, which starts the next operation.

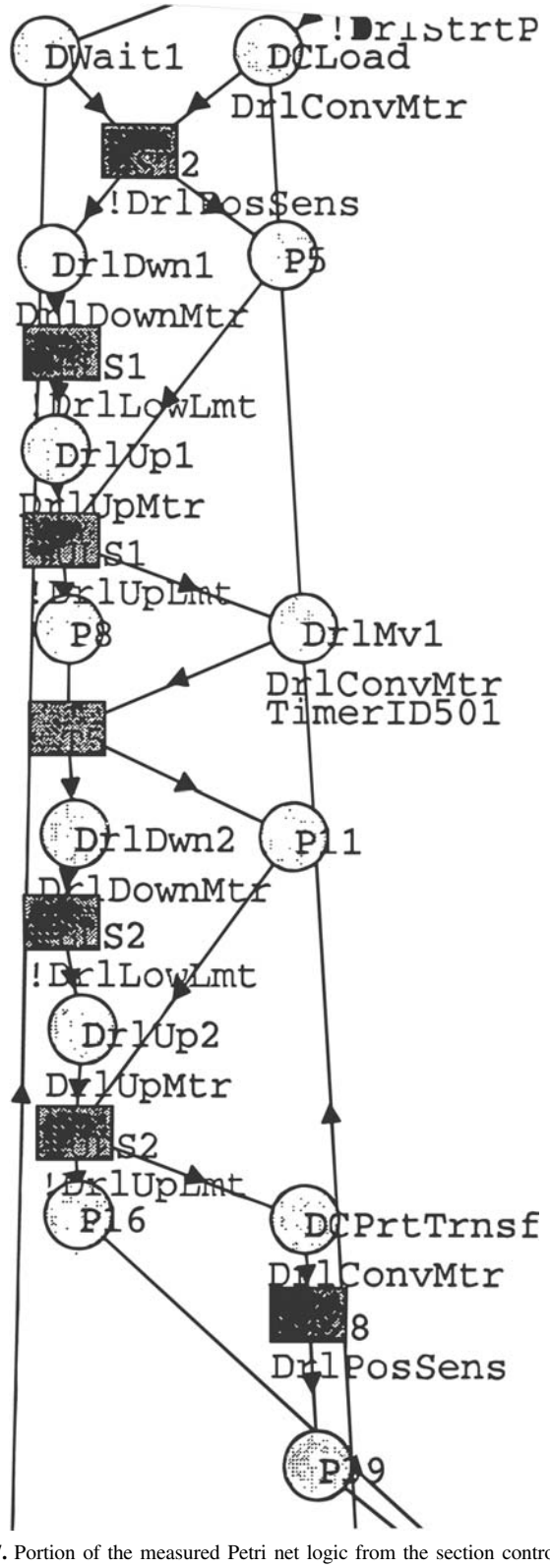

Fig. 7. Portion of the measured Petri net logic from the section controlling the drill station. Items in the left column are used to control the drill spindle and up/down motors; items in the right column are used to control the conveyor. The software used to construct this example was a preliminary academic verison; it was not commercial grade, as can be seen from the poor quality of the user interface.

In this scenario the user must perform a simple mental simulation of the part and the program. However each is a simple sequential simulation and needs only to cover a couple of states. Therefore this problem is moderately difficult.

Scenario 4 (Unexpected system behavior)

As with the ladder diagram, from knowledge of the system sensors it can be determined that if the system is waiting for a part then the introduced part will be treated as the expected one, leaving a part in the system. Because Petri nets only scan expected inputs, if the part is introduced at a sensor when no part is expected it will be ignored by the system until some part is expected from the drill station. In this case the drilled part will not be moved, and will effectively be another unexpected part, now at the drill station.

However, to determine what will happen to the extra part from the Petri net requires a mental simulation of the entire Petri net and the physical system over a large number of interacting states. This problem is hard.

These measurements are summarized in Table 4.

4.4 Analysis of signal interpreted Petri nets (SIPNs)

The signal interpreted Petri net sample was written by Klein [12].

4.4.1 Direct measurements

The signal interpreted Petri net consists of 68 places and 50 transitions. Therefore $N_s = 68$ and $N_o = 50$.

The program is built of one main Petri net with four subnets. Three of the subnets have a single subnet of their own for a total of eight modules. The largest module, which controls a single horizontal milling cycle, contains 10 operations. Therefore $N_m = 8$ and $S = 10/50 = 0.2$.

The average number of causes which must be searched to determine the cause of a transition is 3.66. Most transitions must check one transition connected to its pre-place, one transition connected to its post-place, and a condition on its input. Many require more. The average number of effects which may be the result of a transition is 7.34. This number is greater than the number for standard Petri nets because every output is always explicitly defined in SIPN, whereas the program used in Sect. 4.3 implied that all outputs, which were not turned on, were turned off. If defining an output to be off is not considered an effect, then the average number of effects to be searched is 3.30. Since this definition is closer to that used by other methodologies, we will use $IC_c = 3.66$ and $IC_e = 3.30$.

Table 4. Accessibility summary. Accessibility of information for the scenarios detailed in Sect. 4.2.2. Accessibility is judged subjectively as easy/moderate/hard based on the analysis presented in Sect. 4, with easy being the most useful to a programmer

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Ladder	Easy	Easy[a]	Hard	Hard
Petri net	Moderate	Easy	Moderate	Hard
SIPN	Moderate	Easy	Moderate	Hard
MFSM	Moderate	Hard	Easy	Hard

[a] This problem is easy since it was specifically thought of by the logic designer. Other similar problems are likely harder. See Sect. 4.2.2 for details

4.4.2 Accessibility of data

Scenario 1 (Single output debugging)

There are two modules that control the drilling station of the testbed. These two modules contain a total of 11 transitions and 15 places. Looking over the modules, the first transition in the first module is the only one that turns on the drill conveyor. In addition this transition had no conditions. That makes this scenario seem straightforward.

However, the drill module will not be made active unless all of the modules have correctly completed their respective cycles. Therefore if the drill conveyor will not start, most likely the problem is actually with one of the other modules. Therefore a search of the other modules will be needed to determine where the program has hung up.

Since this requires a search of most of the program, we will judge this scenario to be moderately difficult.

Scenario 2 (System manipulation)

Manipulating the system of signal interpreted Petri nets is very similar to the more typical Petri nets discussed in Sect. 4.3.2. Therefore this problem will be judged easy.

Scenario 3 (Desired system behavior)

The signal interpreted Petri net is laid out such that it is very easy to determine what a particular module will do next. However, as in the typical Petri net, to follow a part through requires somewhat more work. In this case the user must have a minimal knowledge of the physical system to determine what module will the part will enter next, and then must find the correct module to determine what will happen there. Once the correct module has been found, the user will find that there is a module (called "VMill_2"), which defines the sequence Down, Wait, Up, Rotate Tool. This sequence is activated three separate times by the module "VMill_1." This required some searching by the user, and some mental simulation. However, each step was fairly simple.

Since this did not require multiple searches or complex simulations, we judge this moderately difficult.

Scenario 4 (Unexpected system behavior)

As with standard Petri nets, signal interpreted Petri nets only scan for expected inputs. So an unexpected part will have no effect until some part is expected at that location. This will leave an extra part in the system. It is nearly impossible to determine what will happen to that extra part. Doing so requires mentally simulating the entire program and physical machine.

Since this scenario requires a complex mental simulation, we will judge it hard.

4.5 Analysis of a modular finite state machine solution

The modular finite state machines for this study were generated over a period of about four months by an inexperienced under-

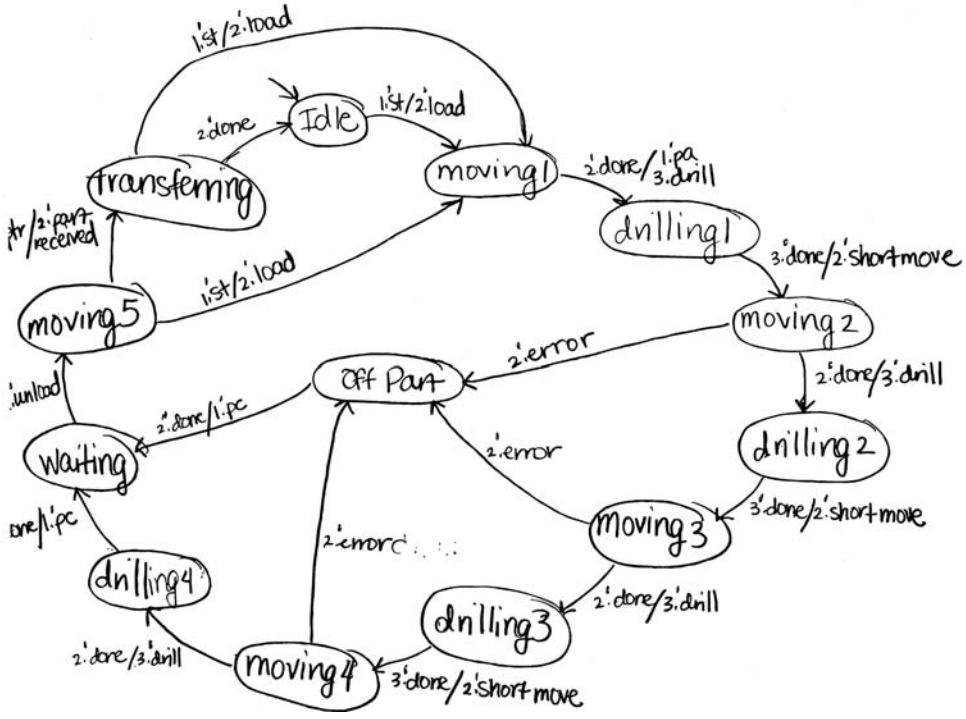

Fig. 8. Portion of the measured modular finite state machine logic. This is the module called `Drill_Controlplan_4holes`; it commands the drill module to drill four holes in sequence (noted by each of the four drilling states). For example, when the system is in the "drilling 1" state, and receives an event "done" from port 3, it sends an event "shortmove" to port 2 and transitions to the "moving 2" state. If a "done" event is received from port 2, then another "drill" command will be sent to port 3 and the system will make a transition to the "drilling 2" state. In addition, some basic error handling has been added to the system, represented here by the transitions surrounding the `Off Part` state. This controller was written by hand with pencil and paper. The software that executes MFSM controllers only has a text-based interface, A full-featured development environment with a graphical user interface is not yet available. The control logic must be typed in by the developer using a text file in a certain format. Even without a sophisticated user interface, the measurement methods described in this paper can be applied to the MFSM logic control design method

graduate working at the University of Michigan with occasional assistance from other members of the research group. The complete logic is contained in [18]. A portion of the logic is shown in Fig. 8.

4.5.1 Direct measurements

The program has 80 states in all of its modules combined and 128 transitions. Therefore the $N_o = 128$ and $N_s = 80$, since all states and operations are directly represented by states and transitions.

The MFSM program has 19 separate modules, which are instances of 13 separate state machines. There are three instances each of a "conveyor coordinator" and "transfer conveyor" and four instances of a "slide." The largest module contains 20 transitions, or 16% of the total. Therefore $N_m = 19$ and $S = 0.16$.

To find the cause or an effect of an operation in MFSMs, a programmer must search through all the transitions in the appropriate connected module. Inputs and outputs are assumed to be a set of one and can be immediately found (i.e., $n_c^i = 1$ if the cause of operation i is known to be an input). Using these criteria the average number of possible causes for an operation is 9.60. The average number of possible effects of an operation is 8.73.

These numbers are summarized in Table 3.

4.5.2 Accessibility of data

Scenario 1 (Single output debugging)
In the MFSM framework, each output is controlled by a single module. To find the reason that the conveyer has not turned on the programmer must look at the state of the module controlling the drill conveyor, which is an instance of the transfer conveyor module. Within this module there are eight states and 20 transitions, and any of the transitions can cause the conveyor to be turned on. Therefore the programmer must determine which state the machine should be in, and which transitions need to occur. Since the states are intelligently named the correct state can generally be found. Then the programmer must determine what event must occur for the correct transition to fire, and if needed follow that event back to the module generating it. In total there are four modules between the start sensor and the command to turn the conveyor on. These four modules contain a total of 31 states and 56 transitions.

While each module has a limited number of states to search though, and intelligent names should help considerably, there are still four possible modules which could be the cause of the problem. This is a considerable search, and therefore this problem is hard.

Scenario 2 (System manipulation)
To manipulate this system back to the idle state all 19 interconnected modules must be manipulated, even those that do not have direct I/O points, but are only controlled through other modules. This will involve the mental simulation of the entire system of 19 modules. This process can be simplified somewhat by manipulating only a few of the modules and trusting the program to manipulate the rest, however this is still not an easy task.

An easier method, if the exact error is known, is to simulate the motion of the removed block through the conveyors. This method utilizes a mental model of the controlled physical sys-

tem, not the program. It will also not work if the exact error is not known.

While generating this program the most common observed method of dealing with this problem was to trip the infrared part sensors haphazardly until either the part system was in the correct state or the user gave up and restarted the program.

Since this involves a complex simulation, this problem is hard.

Scenario 3 (Desired system behavior)

To determine what happens to a part after it has been drilled, a programmer would first need to examine the drill control plan module. That module demonstrates explicitly that the part will be drilled another three times after its first drill. That information is very accessible.

This does not involve any searches or mental simulation, so this problem is easy.

Scenario 4 (Unexpected system behavior)

To determine what will happen if a part is added unexpectedly might involve a full mental simulation of the entire controller/system combination. Most of the information needed can be obtained from the conveyor coordinators, which are three instances of a single conveyor coordinator module. (This module has six states and nine transitions.) However, even with this simplification this requires a mental simulation of three fairly complex modules.

This problem is hard.

4.6 Summary of measurements

Measurements of the sample programs are shown in Tables 3 and Table 4.

The size of the program (measured either by number of states or number of operations) appears inversely related to its modularity. Petri nets are the most simply connected, followed by signal interpreted Petri nets, modular finite state machines, and ladder diagrams. Each framework seems to be good at some scenarios, while poor at others. This agrees with the "match-mismatch hypothesis" [26], which noted that "subjects performed best on 'matched pairs' of tasks and languages."

These measures were, and always must be, based on existing samples of logic. It is possible that as designers become aware of methods that are easier or more difficult to use or debug, that the measures will change. For example, logic written in ladder diagrams can be made somewhat modular by careful design. In addition, careful design can make otherwise difficult scenarios easy, for example scenario 2 in Sect. 4.2.2.

5 Conclusions and future work

The measurements introduced in this paper provide two ways of comparing logic developed in different logic control design methodologies: direct, numerical measures, which provide quantitative measurements of the size, modularity and interconnectedness of logic regardless of which logic control design methodology is used to represent it; and scenario-based measures, which provide a qualitative, user-oriented measure of the effectiveness of a logic control design methodology at representing information.

Based on the measurements of logic samples, it is clear that the method of representation affects the nature of the logic. The ladder representation is smaller, but is very interconnected. The Petri net representations are the least interconnected, although they are significantly larger. The modular finite state machine representation is the most modular, although it is also the largest and is more interconnected than the Petri net based solutions. In addition the difficulty of responding to the different scenarios demonstrates that the method of solution varies significantly across scenarios. These differences will affect the time and cost of developing and maintaining logic.

Although the metrics here provide a basis for comparison, this work does not yet represent a complete understanding of what logic control design methodology is appropriate for a particular application. There are two main areas of future work that will help researchers and practitioners. First, measurements of logic samples that are larger and contain more exception handling will provide more insight into the nature of each logic control design methodology. The samples presented in this paper control a system with 15 inputs and 15 outputs, and contain no exception handling. Industrial scale systems can easily contain 10 000 I/O points, and must handle many error conditions correctly. An important second area of research is to understand the nature of the current logic design process. Understanding the current process will demonstrate the nature of the current problem, and allow future researchers to better match a solution with the problem. This work has begun in [27].

References

1. Lewis RW (2001) Modeling control systems using IEC 61499. The Institution of Electrical Engineers, London
2. Nematron logic control software, http://www.nematron.com/OpenControl/
3. Lewis RW (1988) Programming Industrial Control Systems Using IEC 1131-3 Revised Edition. The Institution of Electrical Engineers
4. Cassandras CG, Lafortune S (1999) Introduction to discrete event systems. Kluwer, Dordrecht
5. Holloway L (2000) Spectool, http://www.crms.engr.uky.edu/pages/spectool/
6. Holloway LE, Guan X, Sundaravadivelu R, Ashley J Jr (2000) Automated synthesis and composition of taskblocks for control of manufacturing systems. IEEE Trans Syst Man Cybern B Cybern 30(5):696–712
7. Park E, Tilbury DM, Khargonekar PP (1999) Modular logic controller for machining systems: Formal representation and performance analysis using Petri nets. IEEE Trans Robot Automat 15(6):1046–1061
8. Park E, Tilbury DM, Khargonekar PP (2001) A modeling and analysis methodology for modular logic controllers of machining systems using Petri net formalism. IEEE Trans Syst Man Cybern C 31(2):168–188
9. Gollapudi C and Tilbury DM (2001) Logic control design and implementation for a machining line test-bed using Petri nets. In: Proceedings of the ASME International Mechanical Engineering Congress and Exposition (Dynamic Systems and Control Division), New York
10. Frey G (2001) SIPN, hierarchical SIPN, and extensions. Technical report, University of Kaiserslautern, Germany, http://www.eit.uni-kl.de/litz/members/frey/PDF/I19.pdf

11. Minas M, Frey G (2002) Visual PLC-programming using signal interpreted Petri nets. In: Proceedings of the American Control Conference, pp 5019–5024
12. Klein S, Weng X, Frey G, Lesage J, Litz L (2002) Controller design for an FMS using signal interpreted Petri nets and SFC: validation of both descriptions via model checking. In: Proceedings of the American Control Conference, pp 4141–4146
13. Klein S, Frey G (2002) Control of a flexible manufacturing system using sipn. Reports of the institute of automatic control i23/2002, University of Kaiserslautern, Germany, http://www.eit.uni-kl.de/litz/members/frey/PDF/I23.pdf
14. Uzam M, Jones AH, Yücel I (2000) Using a Petri-net-based approach for the real-time supervisory control of an experimental manufacturing system. Int J of Adv Manuf Technol, 16:498–515
15. Peng S, Zhou M (2001) Conversion between ladder diagrams and PNs in discrete-event control design — a survey. In: IEEE conference on Systems, Man and Cybernetics, pp 2682–2687
16. Lucas MR, Endsley EW, Tilbury DM (1999) Coordinated logic control for reconfigurable machine tools. In: Proceedings of the American Control Conference, pp 2107–2113
17. Endsley EW, Lucas MR, Tilbury DM (2000) Software tools for verification of modular FSM based logic control for use in reconfigurable machining systems. Japan–USA Symposium on Flexible Automation
18. Shah SS, Endsley EW, Lucas MR, Tilbury DM (2002) Reconfigurable logic control using modular finite state machines: design, verification, implementation, and integrated error handling. In: Proceedings of the American Control Conference, pp 4153–4158
19. Vankatesh K, Zhou M, Caudill RJ (1994) Comparing ladder logic diagrams and Petri nets for sequence controller design through a discrete manufacturing system. IEEE Trans Ind Electron 41(6):611–619
20. Lee JS, Hsu PL (2001) A new approach to evaluate ladder diagrams and Petri nets via the if-then transformation. In: IEEE Conference on Systems, Man and Cybernetics, pp 2711–2716, Tucson, AZ
21. Halstead MH (1977) Elements of software science. Elsevier, Amsterdam
22. Conte SD, Dunsmore HE, Shen VY Software Engineering Metrics and Models. Benjamin/Cummings, Menlo Park, CA
23. Green TR, Petre M (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J Vis Lang Comput 7:131–174
24. Tilbury DM (2001) Logic control testbed, http://www-personal.engin.umich.edu/~tilbury/testbed
25. Tim King Electronics, http://www.phoenix.org/tking/index.shtml
26. Gilmore DJ, Green TR (1984) Comprehension and recall of miniature programs. Int J Man Mach Stud 21:31–48
27. Lucas MR, Tilbury DM (2003) A study of current logic design practices in the automotive industry. Int J Hum Comput Stud 59(5):725–753