Implementation of Discrete-Event Control Solutions

Michael Wood and Karen Rudie

***Abstract*— Despite intensive research on and expansion of the theoretical aspects of Discrete-Event Systems (DES) control theory, a limited amount of research has been reported on its implementation and integration into existing systems. Similarly, a means of identifying which types of systems are good or poor candidates for application of DES control theory and respective methodologies for its implementation require more investigation. We have examined a class of low-level systems for application of DES control theory and proposed a methodology for their implementation. Furthermore, we have developed a software suite with an emphasis on human-computer interaction to facilitate the application of DES control theory in general.**

I. Introduction

DES Control Theory was proposed by Ramadge and Wonham in [1]. A system is modeled as an automaton $\mathscr{G}$ called the plant and is defined as a five tuple $(Q, \Sigma, \delta, q_0, Q_m)$ in the usual way. This means that a human must decide upon a finite set of symbols to compose the alphabet Σ. This decision can be complex and error prone, and it may affect the ease and effectiveness with which the theory can later be applied. Having determined Σ it remains to construct $\mathscr{G}$ such that $L(\mathscr{G})$ describes all behaviour which is possible. Finally, one must define a language based upon Σ that describes behaviour which is not illegal. This can be achieved by defining an automaton $\mathscr{L}$ that generates such a language.

With these constructs, DES control theory can generate an implicit supervisor $\mathscr{S}$, also defined as a five tuple automaton. It is guaranteed that $L(\mathscr{S}/\mathscr{G})$ is a sub-language of $L(\mathscr{G}) \cap L(\mathscr{L})$, which is to say that illegal behaviour is guaranteed to not occur. The term $\mathscr{S}/\mathscr{G}$ indicates the standard closed-loop form. Each element of Σ that is generated by $\mathscr{G}$ is communicated to $\mathscr{S}$ as it is generated, and $\mathscr{S}$ disables or enables each element of Σ before the next event is generated by $\mathscr{G}$.

Discrete-event control theory is based on several primary assumptions, namely, events are generated by the plant, events occur spontaneously, asynchronously and instantaneously and control is imposed by disablement. Many real systems do not easily conform to these assumptions. The problem of real-time arises in the implementation of the closed-loop form $\mathscr{S}/\mathscr{G}$. Certainly the communication of an event from $\mathscr{G}$ to $\mathscr{S}$ may be extremely rapid, but can the enablement or disablement of every element of Σ actually occur before the next event is generated by $\mathscr{G}$? The implementation of the enablement/disablement mechanism can have a devastating impact on the theory.

In a real instance of the classic cat and mouse maze of [2], the events would certainly not occur instantaneously. In other systems, it is inconvenient to model control by disablement alone. Similarly, in systems that are tightly coupled to control objectives, it is difficult to ignore causality, thereby conflicting with the assumption of spontaneity. If one were to first design a pop machine to randomly dispense pop and and then later impose control, the solution would be obviously non-intuitive and inefficient. This exemplifies an inappropriate instance of spontaneity and disablement. In some real systems, with a poorly chosen event set, the theory simply cannot be effectively applied.

It is a fundamental assumption of the framework that the plant exists independent of supervision and control objectives. This is appropriate for systems such as the cat and mouse maze which are composed of various intelligent entities. It is less appropriate for systems such as a vending machine. Once a vending machine is manufactured, there no longer exists a need for control. The question is: "how should it be designed such that it functions in a desirable manner"; consequently, the theory must be applied before the system exists.

The cat and mouse maze is a "good" problem (in the sense of being well suited to DES control theory) because it requires high-level control. It is characterized by an existing system containing entities that can function independent of each other's existence and independent of control objectives.

The vending machine is a "bad" problem because it requires low-level control. It is characterized by entities (such as sensors and actuators) that have no purpose and do not function without control objectives. This coupling requires integration of plant and supervisor in order to produce a reasonable control solution.

II. Tangible Systems

A. Real Systems

The majority of current research in DES control theory has focused on extending the capabilities of the theoretical framework. Certainly this is necessary in the face of state-space explosion and intractable problems. Unfortunately little effort has been focused on how to best use the theory on real problems. Most arguments begin with "Given a plant $\mathscr{G}$...", while very little guidance has been provided on how to best go about defining $\mathscr{G}$ given a real system and a real set of goals. One of the great promises of DES control theory is "correctness by construction", but this requires that the

This work was supported by NSERC and Queen's University

M. Wood was with Queen's University and now works at Entrust. michael@aggressivesoftware.com

K. Rudie is with the Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada. karen.rudie@queensu.ca

plant $\mathscr{G}$ and set of legal requirements $\mathscr{L}$ map accurately to the real system and the real goals. Investigation of the interface between DES control theory and the real world will provide guidance as to which types of systems DES control theory can be advantageously applied, and which systems raise complications and/or render the theory unusable.

Another property of real systems is that they tend to occasionally behave in unplanned ways. What is the impact on the overall behaviour of a system under supervisory control that generates a string not in $L(\mathscr{G})$? Assuming the string contains only elements of Σ, would the event even be communicated to the supervisor? If this behaviour is implementation dependent, what guidelines should one follow when implementing supervisory control? These questions suppose inconsistences in the model, but even with a correct model, the theory can fail.

Consider the cat and mouse maze; not as an analogy for some other system, but a physical maze with a real cat and a real mouse. Can the proposed theoretical solution actually solve this real problem? Consider a general maze; not the one proposed in the classical problem. Presume that the supervisor allows the cat and the mouse to enter non-adjacent rooms R1 and R2 that both have access to a connecting room R3. Let the cat and the mouse simultaneously creep toward their respective entrances to R3. The supervisor cannot handle this situation because it violates the founding assumptions of asynchronous and instantaneous events. A complicated physical realization of semaphores cannot solve this problem because it may violate non-blocking. If R3 is the mouse's only path back to its home room, and if the cat (from R1) can prevent the mouse from entering R3 (via a locking system) then blocking has occurred.

B. Obedient Components

It should be clear that DES control theory was designed and is best suited for high-level decision making control. The cat and mouse maze is an analogy for two intelligent and obedient entities, such as software processes. Both of these properties are important. In a functioning example, the cat cannot simply move from one room to another. It must ask the supervisor if it may perform an action and the supervisor then grants or denies its request. By assuming obedience, we can guarantee asynchronous and instantaneous events. This is exactly the approach taken by the Discrete-event Systems Controller (DESCO) [4] and conflicts with the standard closed-loop model. Instead of having events spontaneously generated by $\mathscr{G}$ and then modifying $\mathscr{G}$ to prevent the future generation of a subset of Σ, we have events spontaneously initiated by $\mathscr{G}$ and their occurrence is allowed or disallowed by $\mathscr{S}$. The obedience of the entities makes the implementation possible via a query/response system instead of the standard closed-loop system. The intelligence of the entities makes the modeling possible, as it accounts for the spontaneous generation of events. Clearly, in a real system, events occur for a reason. The reasons are encapsulated in the intelligence of the components, allowing the DES control theory to view them as spontaneous.

This work was motivated by an interest in the application of DES control theory to small low-level systems with implementations in assembly language programming. Such systems are not characterized by intelligent components. Without intelligence, there is no behaviour and hence no obedience. Since DES constructs are modeled as finite-state machines, and since a pop vending machine is a classical example of a finite-state machine, it seems to be a primary candidate for DES control theory. Unfortunately, its application is not straightforward.

III. Initiated-Event Methodology

A. Overview

Consider a plant with a language over three events labeled **0**, **1**, **2**, and a supervisor with two states *0* and *1*. Assume that the legal constraints on the plant are such that when in state *0*, events **0** and **1** should be disabled, and when in state *1*, only event **2** should be disabled. This can be represented in software as a pointer p and an array 001110, assuming that 1 implies enablement, and the first three digits indicate whether events 0, 1 and 2 should be enabled/disabled at state 0 and the second three digits indicate whether the events should be enabled/disabled at state 1. This software system can embody both the supervisor and the plant. As the plant, it determines that event **1** should be initiated. Next, it jumps to the supervisor subroutine, tests the value of p (which represents the state of the supervisor) and indexes the array at $p \times sizeof(\Sigma) + \mathbf{1}$, assuming indexing starts at zero. With $p = 1$, it would index $001\underline{1}10$ and determine that **1** should be allowed to occur. Before carrying out the occurrence of **1**, it would also have to update p appropriately via a similar data structure. The schema here described illustrates an integration of the plant and supervisor in programmable systems and is a component of what we call the *initiated-event methodology* which is introduced in [3]. Also in [3], concrete examples with a PIC16F84 microcontroller demonstrate beginning–to–end application of the theory including suggestions for automatic machine code generation.

Fundamental to the initiated-event methodology is a particular style of event space definition (namely, abstracting uncontrollable inputs and controllable outputs into a single controllable event that is uncontrollably initiated) and the goal of automatic code generation (for the programmable component of the plant). The initiated-event methodology does not provide a completely automated solution. For each event definition it is required that the conditions for initiation and the work necessary for fulfillment both be provided (in code) by the human designer.

From the point of view of a control solution (such as a microcontroller), the inputs caused by the human users are uncontrollable (values read from input pins) and the outputs generated by the machine are controllable (values written to output pins). This point of view, however, results in a rather unwieldy application of DES control theory. This is why the view must be abstracted. Inputs must be associated with outputs. As the complexity of this association and abstraction increases, the amount of work automated by DES control

theory decreases and the amount of work left to the human and ad hoc methods increases. For this reason it is desirable to model systems with the simplest possible associations between inputs and outputs.

B. A Concrete Example

Consider a fairly high-level view of a very simple vending machine. This vending machine can accept one type of token and dispense one type of pop. The pop costs two tokens. The human users may insert tokens (which may be rejected). The users may also request pop (using a button that may be ignored). The machine contains a maximum of three pop and may be refilled by a human technician.

According to the initiated-event methodology, events should be uncontrollably initiated and be associated with a controllable action. **Pop** could represent the uncontrollable request for a pop paired with the controllable dispensing of a pop. If the supervisor denies the request, the event is considered to not have occurred even though work has occurred within the system. Similarly, **token** could represent the uncontrollable insertion of a token into the machine associated with the controllable rejection of the token. Again, if the token is rejected (which requires work) the event is considered to not have occurred. Finally, **refill** could represent the uncontrollable refilling of the machine. This would be modeled as an uncontrollable event because it is not associated with any output.

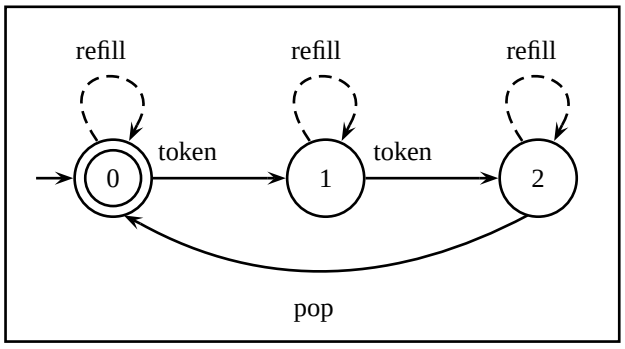

Fig. 1. Rule: pop costs two tokens.

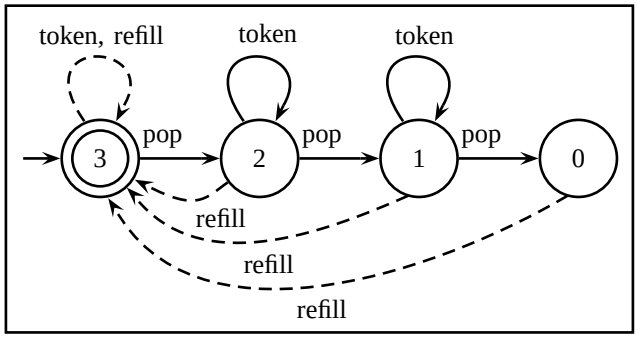

Fig. 2. Rule: the machine should not steal tokens.

The plant could then be represented by the single state automaton with all events in self-loop, and legal behaviour could be captured by two rules: pop costs two tokens (Figure 1), and the machine shouldn't accept tokens when it has no pop to deliver (Figure 2). These rules can be translated into two legal modules (finite-state machines) and the supervisor could be computed. An implementation could represent the plant and supervisor separately in machine code, as in the previous example with events **0**, **1**, **2**.

But the initiated-event methodology can (in some cases) dissolve the separation between plant and supervisor and generate a more intuitive solution. This can be achieved if the lowest level rules have certain properties common to systems such as this. First, it may be the case that there exists a correlation between events from Σ and integer variables that describe the system. In this case we have **tokens** with an initial value of zero that represents the number of tokens received into the machine's bank since the last pop was dispensed. And we have **pops** with an initial value of three that represents the number of pops currently in the vending machine. Note the pluralization of these labels to distinguish them from the corresponding events. The relationships between events and system variables is given in Table I.

Symbol	Impact on tokens	Impact on pops
"token"	tokens = tokens + 1	no change
"pop"	tokens = tokens - 2	pops = pops -1
"refill"	no change	pops = 3

TABLE I

EVENTS IMPACT ON SYSTEM VARIABLES.

It may also be the case that each rule or module of the legal specification has a structure corresponding to exactly one of the system variables. For our system, this is, in fact, the case, and the values of **tokens** and **pops** are given as the state labels of Figures 1 and 2, respectively. If a system has these two properties—(1) mapping between events and system variables, (2) mapping between legal specification state structure and system variables—then the separation between plant and supervisor can be dissolved. Specifically, instead of having each event routine ask the supervisor for permission to execute, each routine could test the system variables against the rules. For this example, the result is more efficient both in runtime complexity and in data complexity than solutions that realize the separate structure of plant and supervisor in assembly language code.

In [3], algorithms are proposed for the automatic generation of assembly language code given only the information contained in Table I and Figures 1 and 2 as input. The output is a complete source code solution lacking only the implementation of the initiating circumstances for each event, and the specific tasks for the enabled and disabled branches of each event. That is, it lacks only the specific realizations of the abstract event definitions.

IV. Software

A. Overview

The majority of DES software, like DES research, is focused on the theoretical and the abstract, such as improved implementations of manipulation algorithms. Little effort has been invested in the development of an intuitive software suite for the purpose of beginning-to-end application of DES control theory. Such a tool would be focused on how best a human might solve a problem using DES control theory with the most automated and least error-prone means possible.

Our Integrated Discrete-Event Systems (IDES) Software was developed to be a robust and usable modeling and pedagogical tool able to output graphs for use in research documents and interface with real systems for demonstration of DES control theory principles. It could serve as a starting point for a standardized and usable conglomerate design tool for many facets of DES control theory. The IDES software accomplishes two goals. First it functions as an interface for specifying DES components in a manner as analogous as possible to pen and paper drawing. Second it demonstrates the integrated use of DES control theory with custom compliant hardware components. It can interface with such hardware via the RS232 protocol.

The IDES software acknowledges the fact that at some point, a human will have to define each of the modules of the plant and legal specification. Should these modules not be defined in software? This implies a need for the manual (but assisted) layout of graphs. Most graph drawing packages focus on the unassisted layout of very large graphs. These algorithms are necessary for DES components generated by DES control theory (such as a supervisor) but not for the initial components themselves. The IDES tool organizes the interaction between computer automation and the human user in a more collaborative manner. This ideology is similar to the approach of the Graph Layout Interactive Diagram Editor (GLIDE) [5], which improves on general constraint-based approaches to graph drawing and layout.

B. Development Process

The development cycle of the IDES interface iterated on experimentation with human users. Subjects ranged from naive (no knowledge of DES, no formal knowledge of graphs, marginal experience with computers) through intermediate (no knowledge of DES, moderate knowledge of graphs, moderate experience with computers) to advanced (considerable experience with DES, formal knowledge of graphs, considerable experience with computers). Users were given a simple graph drawn with pen and paper and a workstation with the IDES software in its initial state and were asked to recreate the graph using the IDES software. This provided unbiased analysis of the usability and effectiveness of the interface. Users were also asked to draw arbitrary graphs with the IDES software as a means of gauging their expectations of its functionality. These experiments lead to the following discoveries.

1) No Logical Relationship Between Graph Components: Naive users consistently persisted in ignoring the relationships between nodes and edges, considering them no more related than lines of ink on paper. This resulted in attempts to create edges before creating nodes.

2) Clicking Paradigms: It was found that regardless of user type, different users unpredictably and approximately equally fell into one of two clicking paradigms. Users would attempt to specify an edge either by mousing down, dragging and releasing or by clicking, moving the mouse and clicking again.

3) Visual Model Stronger Than Mental Model: Many users (including advanced users) failed to choose the internal area of a node as a clicking target for edge termination. Because the arrowhead of an initiated edge tracks the mouse pointer and because completed edges themselves paint from a point on the circumference of a node to a point on the circumference of another node, users would click outside of (but near) a target destination node, instead of inside it.

C. Features

The software provides a standard file system interface, all with intuitive dialogues and safe warning prompts. It also supports export of a selectable “Print Area” to various formats including: “LaTeX”, “EPS”, “GIF” and “PNG”. Furthermore, it fully supports standard editing actions such as “Undo”, “Redo”, “Copy”, “Paste”, “Delete” and “Group Selection”. An intuitive “Zoom” feature is also supported, and all user actions are supported at all zoom levels. For maximum ease of manipulation, the graph canvas can be adjusted both by scrollbars and by a custom “Move” tool.

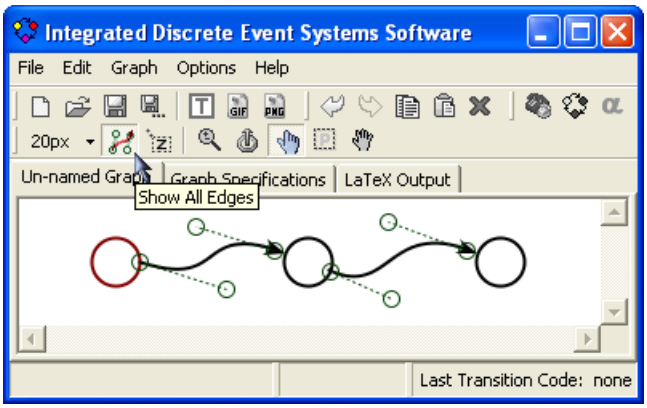

Fig. 3. Edge Manipulation Options.

Three extra items were added to increase the ease of use of the graph drawing functionality. First, a “Grid Options” toolbar item controls the snap-to-grid feature. This applies to the placement of nodes and assists the user in creating nicely aligned graphs. The grid’s visibility can be toggled, and its scale can be modified. Second, a “Show All Edges” toolbar toggles the visibility of the manipulation handles of all edges. This provides a venue for quick custom adjustments to the

layout of the graph. Similarly, a “Show All Labels” toolbar item toggles the visibility of all edge label tethers, resolving any ambiguity between edge label ownership, and providing easy repositioning. Figure 3 shows a customized graph with the edge handles visible.

D. Drawing a Graph

To begin drawing a graph, one must simply click on blank space and a node appears. One may then click on the node to initiate an edge. Next one may either click on another node to terminate the edge, or click on blank space to simultaneously create a new node and terminate the edge. Both paradigms of click—move—click and mouse-down—move—mouse-up are supported. Self-loops can also be created in this way. One can also disconnect an edge from a node and reconnect it elsewhere by clicking on the arrowhead of the edge. Figure 4 demonstrates the addition of a new edge.

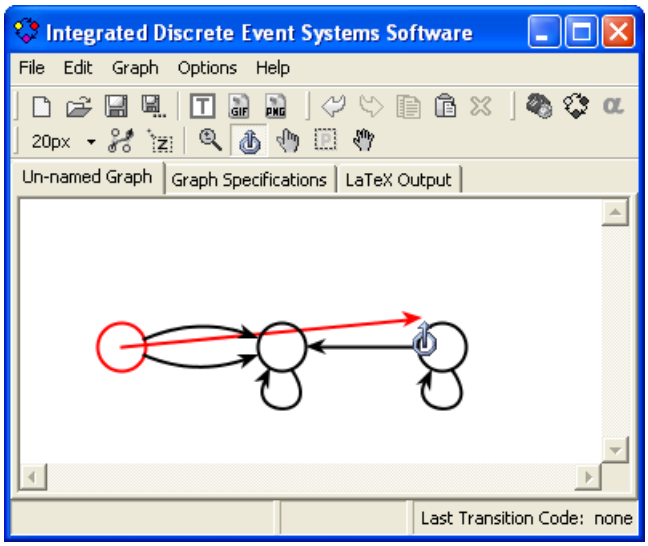

Fig. 4. Drawing Graphs.

Edges will attempt to automatically position themselves in a desirable manner. When the default behaviour would cause overlap or collision, the edges attempt to reposition themselves accordingly, and always attempt to display themselves in a symmetrical and reasonable manner. These automatic positioning algorithms are based on a set of common configurations. Since the default layout algorithms are not perfect, the user will need to customize the position of elements in the graph. The “Modify Nodes, Edges or Labels” tool facilitates this. The user can mouse-down on a node, drag it around, and mouse-up to reposition it. All the edges connected to the node will adjust with the user’s movement. To adjust edges the user must first click on their arrowheads. This selects the edge and causes it to draw its anchors (four green circles). The user may click on and drag the anchors to reposition them. The edge will update accordingly.

The modify tool can also be used to group objects. This is done in the standard way supporting both CTRL-click and mouse-down—move—mouse-up to specify a bounding box. The user can then mouse-down, drag and release to move all the objects in the selected group. External edges connected to the group will update in the usual way, attempting to maintain their specified configurations. Several right-click popup menus are also provided. These support standard actions based on the context of the right-click. Such actions include “Snap To Grid”, “Reset Configuration”, “Copy”, “Paste”, “Delete”, “Undo”, “Redo”, “Label”, “Initial State” and “Marked State”.

To add or edit the label for a node, the user may simply double-click the node when the “modify tool” is selected. This opens a popup window where the user may type the text for the node. When the user is done, they may simply click back anywhere on the main window, or hit ENTER (CTRL + ENTER achieves a new line). Optionally, labels can either be standard text or LaTeX code. When working with LaTeX, the source is shown in the input box and the rendered result is shown on the graph.

The alphabet Σ for edge labels is specified in a separate window, and each event can be associated with a variety of meta-data. To add a label to an edge, the user must simply double-click on the edge’s arrowhead or existing label. This raises a transition chooser popup which is simply an array of toggle buttons; as the user switches them on and off, the respective values appear or disappear near the edge. Events appear as a comma-delimited list in the order in which they are added to the edge. Any edge that is associated with at least one uncontrollable transition is drawn as a dashed, rather than a solid, line.

E. Animated Trace

This feature was developed for use with custom external hardware representing a real plant. The external hardware was designed to transmit single bytes representing events as they occurred in the plant. In the “Graph Specifications” tab, each event may be assigned a value in the “machine code” field. The “machine code” field specifies which bytes map to which events in the graph model. The “Start Trace” toolbar item allows the user to initiate an animated trace of transitions in the graph. The system attempts to establish communication over the COM1 port of the user’s computer (9600 baud, 8 data bits, no parity, 1 stop bit, no flow control). When a trace is started, the initial state of the graph becomes highlighted in blue. The system then listens for transitions. When they occur, the blue highlight animates across that transition to the appropriate state, as shown in Figure 5.

The software and machine communication is based on single byte packets which are interpreted as integer values between 0 and 255. This limits the effective event space to 256 elements, but was deemed sufficient for simple modeling purposes. As an added feature, the system can also be used as a means of control. When a byte is received from COM1 and an associated outgoing transition is found, the software sends the same byte back out COM1, but if no matching transition is found, it does not echo the byte. Custom hardware can interpret this as control. If the custom hardware consistently sends controllable event codes before the events occur, it

can interpret the lack of an echo as indication that the event should be disabled.

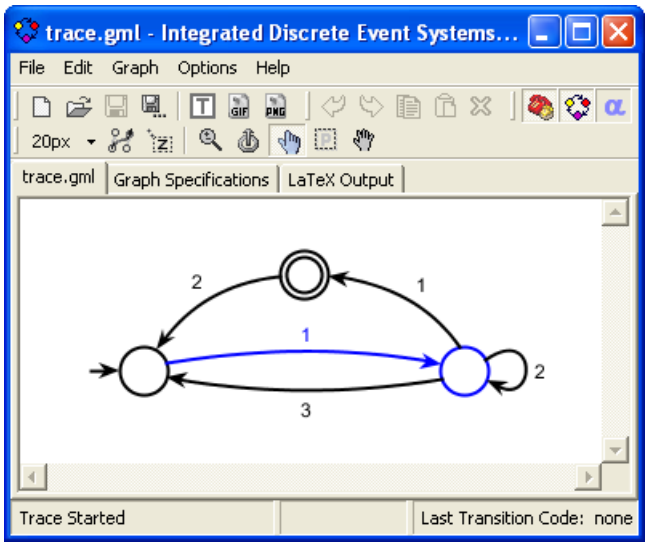

Fig. 5. Trace of a Real System.

Figure 6 is a photograph of a real model that functioned under DES control imposed by the IDES software. It contains two microcontrollers, a small LCD and several push buttons and LEDs. These components represent the behaviour of a simple vending machine. The code on the microcontroller notifies the IDES software of various events and listens for an echo of its notification. Without this echo, it assumes disablement. By running an appropriate supervisor in the IDES software connected to the hardware component, the occurrence of certain events (initiated by a human pressing a button) are prevented.

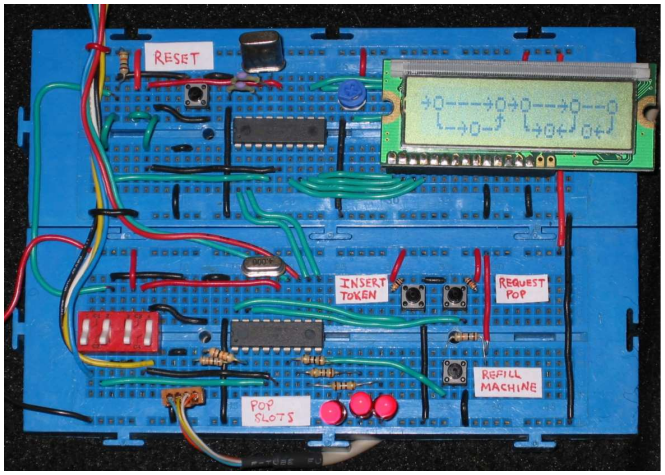

Fig. 6. A Real System Used In Experimentation.

V. CONCLUSIONS AND FUTURE WORK

A. Conclusions

While advances continue to be made in the theoretical framework of DES control, insufficient energy has been focused on investigating its implementation and integration into new and existing systems. We lack a means of identifying which types of systems are good or poor candidates for application of DES control theory, and corresponding methodologies for its implementation require more investigation. We have examined a class of low-level systems for application of DES control theory and proposed a methodology for system implementation. The initiated-event methodology is both an ideology for the application of the modeling theory and a framework for increased automation in the generation of a final control solution. We have also developed a software suite with an emphasis on human-computer interaction to facilitate the application of DES control theory in general. The IDES software minimizes human labour in the design of DES components, and interfaces with custom hardware to demonstrate DES control.

B. Future Work

The IDES Software is incomplete. To be truly useful as a design tool, it must interface with DES algorithm toolkits to allow operations to be performed on the input models. Work along these lines is currently being carried out in the DES Lab at Queen's University. Furthermore, for a real beginning-to-end solution for application of DES control theory to low-level microcontroller systems, it would beneficial to implement the automatic code generation for the initiated-event methodology suggested in [3].

VI. ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC). We thank Lenko Grigorov for his suggestions, which improved our software

REFERENCES

[1] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete-event processes", *SIAM Journal of Control and Optimization*, vol. 25, num. 1, 1987, pp 206-230.

[2] W. M. Wonham and P. J. Ramadge, "On the Supremal Controllable Sublanguage of a Given Language", *SIAM Journal of Control and Optimization*, vol. 25, num. 3, 1987, pp 637-659.

[3] M. M. Wood, "Application, Implementation and Integration of Discrete-Event Systems Control Theory", Master's Thesis, Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, 2005.

[4] M. Fabian and A. Hellgren, "DESCO—a tool for education and control of discrete event systems", Discrete Event Systems, Analysis and Control, 2000, pp 471472.

[5] K. Ryall, J. Marks and S. Shieber, "An interactive constraint-based system for graph drawings", In the Proceedings of the 10th Annual Symposium on User Interface Software and Technology, Banff, Alberta, 1997, pp 97-104.