2008 American Control Conference
Westin Seattle Hotel, Seattle, Washington, USA
June 11-13, 2008

Exception Handling Controllers: An Application of Pushdown Systems to Discrete Event Control

Christopher Griffin
Oak Ridge National Laboratory
P.O. Box 2008 MS 6418
Oak Ridge, TN 37831
E-mail: cgriffin229@yahoo.com

***Abstract*— Recent work by the author has extended the Supervisory Control Theory to include the class of control languages defined by pushdown machines. A pushdown machine is a finite state machine extended by an infinite stack memory. In this paper, we define a specific type of deterministic pushdown machine that is particularly useful as a discrete event controller.**

Checking controllability of pushdown machines requires computing the complement of the controller machine. We show that Exception Handling Controllers have the property that algorithms for taking their complements and determining their prefix closures are nearly identical to the algorithms available for finite state machines. Further, they exhibit an important property that makes checking for controllability extremely simple. Hence, they maintain the simplicity of the finite state machine, while providing the extra power associated with a pushdown stack memory. We provide an example of a useful control specification that cannot be implemented using a finite state machine, but can be implemented using an Exception Handling Controller[1].

I. Introduction and Background

The supervisory control theory (SCT) for discrete event dynamic systems (DEDS) was introduced by Ramadge and Wonham [1]. The initial work on regular languages was extended in many subsequent publications. Examples of these extensions can be found in [2]–[14]. In SCT a plant model is given by a finite state machine $G = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$. Here, Q is a finite set of states; Σ is the finite event alphabet with $\Sigma = \Sigma_c \cup \Sigma_u$; q_0 is the start state; Q_f are the final states and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. The events in Σ_c may be disabled by a supervisor and hence are called controllable. A second machine, M_K may be run in parallel with the plant G, thus controlling the stream of symbols generated by G. A complete description of SCT may be found in [1] or [15].

[12] extends the SCT to the case when the controller machine M_K is given by a deterministic pushdown machine (DPDM). Informally, a pushdown machine (PDM) is a finite state machine augmented with an infinite stack memory. Formally, A pushdown machine is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f \rangle$, where Γ is a stack alphabet, Z_0 is the initial symbol on the stack and $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma \times Q \times \Gamma^*$ is the transition relation in the PDM.

[1]A portion of this work was created as a Eugene P. Wigner Fellow and staff member at the Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy under Contract DE-AC05-00OR22725.

It is sometimes convenient to think of δ as a mapping from $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. In this case we will write $\delta(q, a, Z)$ to denote the finite subset of $Q \times \Gamma^*$ that results. Both notations will be used throughout this paper and we will choose which notation to use based on convenience and clarity of expression.

A PDM consists of an *input tape*, a *finite state system*, and a *stack*. Symbols are recognized from the input tape *one-by-one*. Based on the top stack symbol, the input symbol and the current state of the machine, a (*possibly non-deterministic*) transition will be made that changes the state of the finite state system, and replaces the top stack symbol with a (*possibly empty*) string of stack symbols. (See Figure 1.)

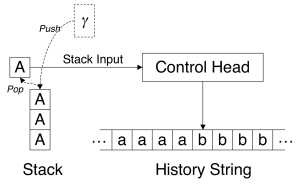

Fig. 1. The operation of a pushdown machine.

Let M be a PDM. By $\delta(q, a, Z) = \{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ we mean that given an input symbol a when M is in state q with top stack symbol Z, then M may transition to a new state p_i and replace Z with the string γ_i for $1 \leq i \leq n$.

Under some conditions, the PDM can edit its top stack symbol independent of input events. This occurs when an ϵ-transition is fired. In this case, both the state and top stack symbol of the PDM may be altered, but no new symbols from the input string are accepted. By $\delta(q, \epsilon, Z) = \{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ we mean that when M is in state q with top stack symbol Z, there may be an ϵ-transition which, independent of any input symbol, causes M to transition to a new state p_i and replace Z with γ_i for some $1 \leq i \leq n$.

A PDM M is *deterministic* (DPDM) if the following hold:

1) If $(q_1, a, Z, q_2, \gamma) \in \delta$ and $(q_1, a, Z, q_2', \gamma') \in \delta$, then $q_2 = q_2'$ and $\gamma = \gamma'$.
2) If $(q_1, \epsilon, Z, q_2, \gamma) \in \delta$, then for all $a \in \Sigma$, $\delta(q_1, a, Z) = \emptyset$.

Unlike finite state automata, the state of a PDA is captured by both the state of the machine and the value stored in the

stack. To formally describe the state of a PDA, we must specify these values. Traditionally the remaining input string to be read is also included in this description [16].

An *instantaneous description* (ID) for a PDA M is a triple (q, w, γ), where $q \in Q$ is the state of M, $w \in \Sigma^*$ is the remainder of the string to be read and $\gamma \in \Gamma^*$ is the state of the stack.

Let M be a PDA. By $(q, aw, Z\gamma) \vdash (p, w, \beta\gamma)$, we mean that $\delta(q, a, Z)$ contains (p, β). Let $\vdash^*$ be the transitive closure of $\vdash$. Then by $(q, wv, \gamma) \vdash^* (p, v, \beta)$, we mean that there are a series of transitions in δ that causes M to read the prefix w and transition from state q to state p and transform the stack string from γ to β. The remaining string v is still available for reading.

By $\mathcal{L}_{\mathrm{M}}(M)$ we mean the set of strings (language) that takes a PDM from its start state/stack configuration and ends in a final state (and arbitrary stack configuration). If $L = \mathcal{L}_{\mathrm{M}}(M)$, then we say that L is context free–or a context free language (CFL)–when M is a PDM. The language L is deterministic context free–or a deterministic context free language (DCFL)–when M is a DPDM. The term $\mathcal{L}_{\mathrm{M}}(G)$ may be defined analogously when G is a finite state machine. See [15] for details. If a language is generated by a FSM, then we say it is a regular language [16].

Let $L = \mathcal{L}_{\mathrm{M}}(G)$. We denote the prefix closure of L by $\overline{L}$ and say that a language is prefix closed if $L = \overline{L}$. Details of these terms may be found in [1], [12], [15]. Let M_K be a controller machine and let G be a plant model if $K = \mathcal{L}_{\mathrm{M}}(M_K)$ and $L = \mathcal{L}_{\mathrm{M}}(G)$, we say that K is controllable with respect to G if $\overline{K}\Sigma_u^* \cap \overline{L} = \overline{K}$ [4]. The concept of controllability is central to the study of the SCT.

Two key properties of both the regular and deterministic context free languages are (i) closure with respect to complementation (with respect to Σ^*) and (ii) closure under prefix-closure. Further if M_K is a DPDM and G is an FSM, then $\mathcal{L}_{\mathrm{M}}(M_K) \cap \mathcal{L}_{\mathrm{M}}(G)$ is effectively generated by another DPDM.

In [12] we proved that the statement K *is controllable with respect to* L is decidable when K is generated by a DPDM M_K and L is generated by an FSM and $K \subseteq L$. To show this, we proved that the following statements were equivalent:

1) $\overline{K}\Sigma_u^* \cap \overline{L} = \overline{K}$.
2) $\left(\left(\overline{K}\right)^c \cap \overline{L}\right) / \Sigma_u^* = \left(\overline{K}\right)^c \cap \overline{L}$.

We then showed that if (2) above holds if and only if when N is a DPDM accepting $\left(\overline{K}\right)^c \cap \overline{L}$, then there is no string su with $s \in \Sigma^*$ and $u \in \Sigma_u^*$ such that

1) s is a prefix of a string in $\left(\overline{K}\right)^c \cap \overline{L}$ taking N to some none-final state
2) u will force N to transition from this non-final state to a final state.

This property can be verified algorithmically. We called it being closed under Σ_u reverse paths [12]. The key property of the Exception Handling Control structure we will define is that this property–being closed under Σ_u reverse paths is very easy to check. This makes Exception Handling Controllers attractive as DPDM discrete event control systems.

The remainder of this paper is organized as follows: In Section II we define the Exception Handling Controller (EHC) structure. In Section III we prove the formal properties of EHC and show that checking controllability is simpler than for arbitrary DPDM. We show that they are highly similar to FSM controllers. In Section IV we provide an EHC example that is strictly more powerful than any FSM controller. By this, we mean that we present a control system whose logic and resulting behavior can only be produced when a pushdown machine is used as the controller. We present conclusions in Section VII.

II. Exception Handling Controllers

An Exception Handling Controller (EHC) is a DPDM with the following state types:

1) Execution States (required),
2) Response States (required),
3) Stack Modification States (optional),
4) A single Start State (required) and
5) A single Stop State (optional).

If no stop state is given, the controller must execute indefinitely.

For each controllable event $c \in \Sigma_c$ there is a corresponding stack symbol $C \in \Gamma$. Additionally, there may be a finite, but arbitrarily large, number of auxiliary stack symbols $Z_0, \ldots, Z_n$. Then $\Gamma = \bigcup_i \{Z_i\} \cup \bigcup_{c \in \Sigma_c} \{C\}$. The transition structure of an EHC has the following properties

1) If q_0 is a start state, then there is a single transition of the form $(q_0, \epsilon, q_E, Z_0, \gamma Z_0) \in \delta$ for some executor state q_E.
2) If q_E is an executor state, then for some (possibly empty) set of controllable events $E \subseteq \Sigma_c$, $\delta(q_E, c, C) = (q'_E, \epsilon)$ or $\delta(q_E, c, C) = (q'_E, C')$ for some other executor state (or the Stop State) q'_E and for each $c \in E$.
3) If q_E is an executor state, then for some $u \in \Sigma_u$ $\delta(q_E, u, Z)$ is defined for some $Z \in \Gamma$ and further, $\delta(q_E, u, Z) = (q_R, Z)$ where q_R is a response state. Further, for any Z_i, if $(q_E, \epsilon, Z_i, q'_E, \gamma) \in \delta$ is defined, then there are no other transitions from q_E with top stack symbol Z_i and q_E is an executor state.
4) If q_R is a response state then there is at least one transition of the form $(q_R, \epsilon, Z, q', \gamma) \in \delta$ where q' is a stack modification state *or* an execution state.
5) If q_M is a stack modification state, then there is at least one transition of there form $(q_M, \epsilon, Z, q', \gamma) \in \delta$ for some $Z \in \Gamma$ where q' is either *different* stack modification state *or* an execution state.
6) If q_S is a stop state, then there are a set of looping ϵ-transitions at q_S that empties the stack.

In an EHC, every executor state is final and Z_0 is the initial stack symbol.

Figure 2 shows the general structure of an EHC.

Remark 1: We will implicitly assume that the ϵ-transitions associated with the stack modification states produce a

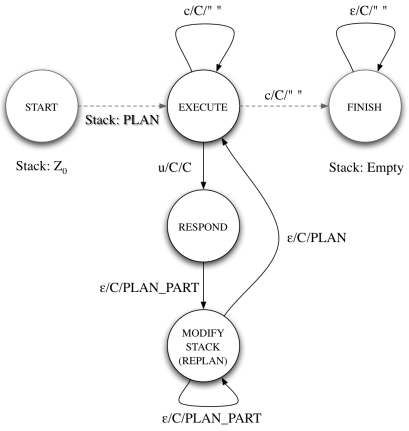

Fig. 2. Exception Handling Controller–General Structure

consistent path to return to an execution state. That is, we assume that there is no chance of deadlocking in one of these states because of improperly formed stack operations. This fact is easily checkable by verifying that the push and pop operators defined in the ϵ-transitions provide a path entering and leaving these states and by ensuring that if the top stack symbol is unknown, all possible top stack symbols are considered in the transitions.

A. Types of Stack Modification Operations

Stack modification occurs in the stack modification states and is in response to uncontrollable events. (It is possible to use the stack markers $Z_0, \ldots, Z_n$ to force re-planning as well using an ϵ-transition from the execution state however, we will not discuss this in detail.) There are three types of modification that can occur in response to an uncontrollable event:

1) Erasure: In an erasure operation, part of the existing stack is irretrievably erased. The "planned" controllable events corresponding to this portion of the stack may not be executed.
2) Write: In this case, new stack symbols are written directly above the existing stack symbols. Any plans currently in operation are suspended and a new plan begins execution.
3) Shuffle Write: Using finite state storage, a portion (whose length is proportional to the amount of finite state storage available) is removed from the stack. A new stack is appended below this plan and then the previously executing plan is rewritten to the stack.

We give an example of conditions where each of these operations is rational:

1) **Erasure** The controller is directing the top-level activities of an air campaign. The uncontrollable event indicates that a target is no longer of importance. Existing plans to attack it are erased.
2) **Write** The controller is directing high-level behavior of a robotic rover. A message is received from an operator indicating that a soil sample must be taken at present coordinates immediately. All plans currently in operation are suspended and a new plan is executed.
3) **Shuffle** The controller is directing the behavior of a manufacturing system. Requests must be kept in relative order, but all requests must be processed (hence infinite storage memory may be required to avoid state explosion). New orders are placed as far down on the stack as possible to simulate a FIFO queue.

Note in the Shuffle example, a true FIFO queue *cannot* be simulated with a single pushdown stack and a finite number of states. Two pushdown stacks are required. A machine with two pushdown stacks is equivalent to a Turing machine and hence by the undecidability results given in [12], arbitrary control systems of this type cannot be verified for controllability.

III. Properties of Exception Handling Controllers

Proposition 2: If M is an exception handling controller accepting $\mathcal{L}_{\mathrm{M}}(M)$, then $\mathcal{L}_{\mathrm{M}}(M)$ is prefix closed.

Proof: Let $w \in \mathcal{L}_{\mathrm{M}}(M)$. By definition, the only accepting states of M are execution states. Therefore, there is a series of ϵ transitions taking w from some state that is not an execution state to a state that is an execution state. Suppose that $w = sa$, where $a \in \Sigma$. Suppose that $(q_0, s, Z_0) \vdash^* (q, \epsilon, \gamma)$. If q is an execution state, then s is accepted. Suppose that q is not an execution state. If it is a response state, then there is a series of ϵ-transitions leading back to an accepting state, and hence s is accepted. If, q is a stop state, then there is no string extending s and hence w does not exist. By assumption, we know that if q is a stack modification state, then there is a series of ϵ-transitions leading back to an accepting execution state. Hence s is accepted. Trivially, $\epsilon \in \mathcal{L}_{\mathrm{M}}(M)$ by the definition of exception handling controller. This completes the proof. ■

Proposition 3: If M is an exception handling controller, then a machine M^c accepting $\mathcal{L}_{\mathrm{M}}(M)^c$ is derived from M by:

1) Adding a single state (DUMP).
2) If q_E is an execution state and $a \in \Sigma$ is not defined for a stack symbol $A \in \Gamma$, then define $(q_E, a, A, \mathrm{DUMP}, A) \in \delta$, the transition function.
3) For each execution state q_E declare q_E non-final and
4) Declare DUMP final.
5) For all $a \in \Sigma$, $A \in \Gamma$ define $(\mathrm{DUMP}, a, A, \mathrm{DUMP}, A) \in \delta$.

Proof: Let $L = \mathcal{L}_{\mathrm{M}}(M)$. We will show $L^c \subseteq \mathcal{L}_{\mathrm{M}}(M^c)$. Choose $w \in L^c$. There is a shortest prefix of w such that $w \in L$. Let $w = st$, where s is this prefix. We have already shown that L is prefix closed and further that any accepted string must lead to an execution state. Since the structure of M^c is identical to that of M except when M is undefined at execution states, it follows that $(q_0, s, Z_0) \vdash^* (q_E, \epsilon, \gamma)$ where q_E is an execution state. Let $t = av$, $a \in \Sigma$ and let $\gamma = A\gamma'$. Trivially, if a transition at a were defined at q_E with

stack symbol A, then s would not be the shortest prefix of w accepted by M. Hence, when reading w with M^c, there will be a transition to DUMP on a. The definition of the transitions at the dump state ensures that w will be accepted by M^c. Hence $L^c \subseteq \mathcal{L}_{\mathrm{M}}(M^c)$.

We will now show that $\mathcal{L}_{\mathrm{M}}(M^c) \subseteq L^c$. Choose $w \in \mathcal{L}_{\mathrm{M}}(M^c)$. By definition, w is accepted at DUMP and there is a longest prefix $w = st$ such that $s \in L$; without loss of generality, let $t = a$, $a \in \Sigma$. If $sa \in L$, then there is a transition in M at some state q_E defined for a and an appropriate stack symbol. Clearly, this contradicts our definition for M^c, which is still deterministic. Thus, it follows that $w \notin L$. Thus we have shown that $\mathcal{L}(M^c) = L^c$. ■

Definition 4: A *quasi-exception handling controller* is a deterministic pushdown machine with all the properties of an exception handling controller *except* not all execution states need be final.

Proposition 5: If M is an EHC accepting $S = \mathcal{L}_{\mathrm{M}}(M)$ and G is a finite state machine (with no non-deterministic) transitions accepting $L = \mathcal{L}_{\mathrm{M}}(G)$. Then $S \cap L$ is accepted by a quasi-exception handling controller.

Proof: The proof is a straight-forward consequence of intersecting a deterministic pushdown machine and a finite state machine. (See Chapter 2, Section 2.7.6.) ■

Remark 6: The prefix closure of a quasi-exception handling controller is obtained by marking every execution state. Hence, this fact combined with Propositions 3 and 5 will be important for efficiently computing the controllability predicate.

Furthermore, taking complements in quasi-exception handling controllers is precisely like taking complements in exception handling controllers except we reverse the marking on the execution states; i.e., some execution states may be non-final. These become final in the complement.

Proposition 7: Let G be a plant model, M an exception handling controller and M_K the result of intersecting M and G. Let $K = \mathcal{L}_{\mathrm{M}}(M_K)$ and $L = \mathcal{L}_{\mathrm{M}}(G)$. Let N be a machine accepting $\overline{L} \cap \overline{K}^c$. Then K is controllable with respect to L if and only if there is no uncontrollable transition leading from a non-final state of N to a final state of N.

Proof: ($\Rightarrow$): Necessity is a trivial consequence of Proposition 4.2 of [12].

($\Leftarrow$): Sufficiency is established by showing that whenever M_K is uncontrollable there is an uncontrollable transition connecting a non-final state of N to a final state of N by an uncontrollable transition. By Proposition 3 when computing the complement of an EHC, it follows that no ϵ-transitions will lead from the non-final states to the final states. Intersecting such a complement with a finite state machine will not change this fact. Thus, it follows at once that if there is any Σ_u reverse path in N, it must be a single uncontrollable transition. ■

IV. Example

We consider a contrived automated fabrication scenario. A machine has been constructed that can take requests from users who require widgets. The users and machine are separated, so user requests may come into the machine while it is building a widget. Manufacturing of a *widget* requires two basic steps:

1) Cut (the basic shape),
2) Bend (make a bend in the shape),

The machine controlling the cutting/bending must record each time a request is generated so that the appropriate number of widgets is produced. With no a priori information on the number of widgets that could be requested, the system cannot keep track of the correct number of widgets to make with a finite number of states. The stack of an EHC can be used to keep track of this information. When the machine has been inactive for a period of time, it returns to a sleep mode. The plant model for this machine is shown in Figure 3.

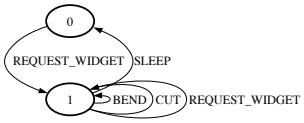

Fig. 3. Simple plant model for widget making machine.

We will define three stack symbols for the EHC:

1) Z_0: The initial stack symbol–indicates that no widgets need to be made currently.
2) C: The cut symbol–indicates a cut event should be generated.
3) B: The bend symbol–indicates a bend event should be generated.

A proposed EHC controller is shown in Figure 4 The

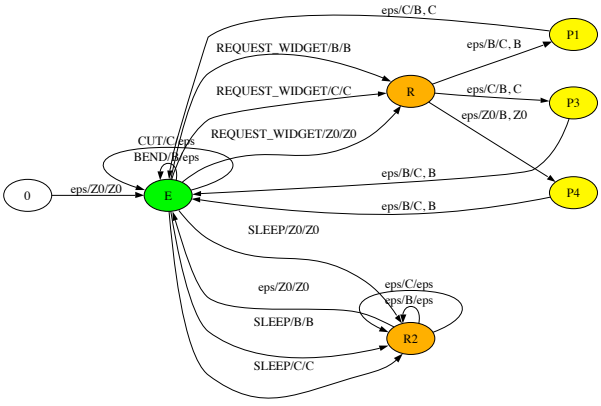

Fig. 4. Simple controller model for widget making machine.

possible stack strings of this controller will be either:

1) $(CB)^+Z_0$: In this case, some number of widgets is to be made.
2) $B(CB)^*Z_0$: In this case a bend must be completed to finish making a widget and other widgets may need to be made.
3) Z_0: No widgets are to be made.

The system begins in State E. When widget requests come in, the system transitions to State R and the stack is modified to place the appropriate commands on to it. Stack modification occurs in states $P1$, $P3$ and $P4$, depending upon the current stack configuration. The system returns to E and widget production continues. If a sleep event occurs, the system clears its stack to Z_0 and returns to the execution state to await more widget commands. We assume, in normal functioning, that the sleep command will only occur when the system has stack Z_0 for otherwise it would be in the middle of widget processing.

Checking controllability of this system can be accomplished by analyzing the machine shown in Figure 5. The

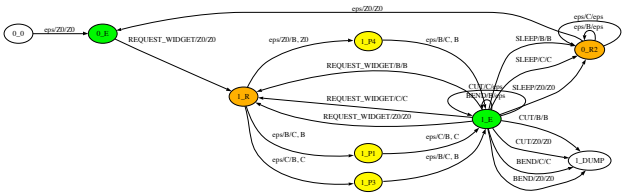

Fig. 5. A machine accepting $\left(\overline{K}\right)^c \cap \overline{L}$ when L is the plant language and K is controller language. Note it has no uncontrollable transitions leaving entering the non-dump states.

dump states are the final states of this system. Note there are no uncontrollable transitions entering the dump states. Hence, it follows from Proposition 7 that the supervisory language generated by the EHC must be controllable with respect to the plant language.

A. Discussion of Example

The control system we developed in the previous section has a capability at cannot be modeled using a finite state system: *The machine controlling the cutting/bending must record each time a request is generated so that the appropriate number of widgets is produced.* This control requirement implies that the number of requested widgets must be recorded in real-time by the machine controlling widget construction. The plant model specifies that a widget may be requested at anytime. That is, the total number of widgets requested in the system may change as time moves forward and is not pre-specified by the structure of the plant. The controller must be able to store an arbitrary and unspecified number of widget requests. This immediately pre-suppose the existence of non-finite memory, which is not possible with a finite state machine plant and controller combination. It is possible, as we have shown, using an EHC controller and finite state plant.

V. Extended Example

We can extend the example provided in the previous section. Suppose that after each action (bend or cut) an eye evaluates the product and determines whether it is acceptable to continue or not. If it is not acceptable, an error is generated and the widget making machine must be cleared and reset. The new plant is shown in Figure 6. In this model State 3 is

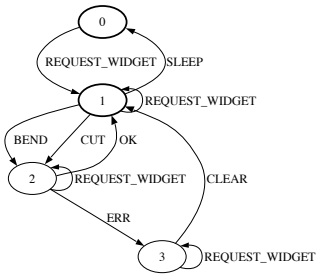

Fig. 6. A plant model with error detection.

an error state in the plant. The EHC defined in the previous section can be extended to include new response states to deal with the uncontrollable events OK and ERR. Response states $R3$ and $R4$ properly modify the stack to execute an appropriate clear command on the plant. The new controller is shown in Figure 7. Checking controllability is similar to

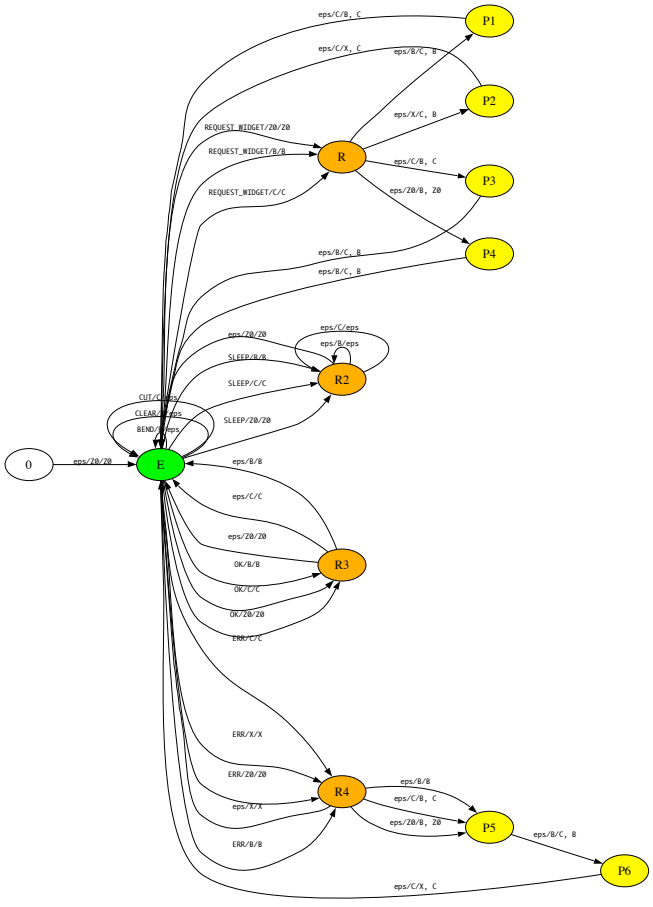

Fig. 7. A new controller taking into account errors in the production process.

the example in the preceeding section, however it is clear from the example that the system will be controllable.

VI. Limitations of the Approach

This example could be extended further by modeling a plant with multiple modes of execution, leading to multiple

types of executor states within the controller. In this case the example could have two types of widgets required for production and use more controllable events. Such an example would illustrate the limitations of exception handling control. Consider a control system that produces two types of widgets. As new widget requests are made they are stored on the stack. If the example is extended in a straight-forward way, however, the order of the requests is reversed. Hence, widget requests are satisfied in the opposite order of their occurrence. We can use additional stack modification states to store a component of the current stack (and thus preserve order), but again we will run into a finite memory problem. Since some of the stack information must be transferred to the finite memory component of the controller (i.e., the states) only a finite order can be maintained. Hence for some fixed N widget requests we can maintain an appropriate order, but the $N + 1^{\text{st}}$ widget may not be in the correct order in which it was requested. A control system that is capable of storing an arbitrary request order for multiple widget types would require a queue storage system. Using the results in [12], we can see that there is no algorithm for checking controllability in such a system. It should be noted that meta-proofs using induction can be used to prove controllability for *some* systems like this, but not all [17].

VII. Conclusion

In this chapter we have defined the concept of Exception Handling Controller. Exception handling controllers are deterministic pushdown machines that are designed to store an executable plan on the pushdown stack. They have computational properties that make them similar to, but more powerful than, finite state machines. We provided a very simple yet illustrative example of the use of EHC machines in a manufacturing scenario. One major benefit to the use of a pushdown stack memory store in discrete event control is the ability to store a plan. This could allow hierarchical or hybrid controllers to inspect the contents of the pushdown stack. This is a novel approach to hierarchical discrete event control and one that may be worthy of further study.

References

[1] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control Optim.*, vol. 25, no. 1, pp. 206–230, 1987.

[2] ——, "On the supremal controllable sublanguage of a given language," *SIAM J. Control Optim.*, vol. 25, no. 3, pp. 637–659, 1987.

[3] R. D. Brandt, V. K. Garg, R. Kumar, F. Lin, S. I. Marcus, and W. M. Wonham, "Formulas for calculating supremal controllable and normal sublanguages," *Systems Control Lett.*, vol. 15, pp. 111–117, 1990.

[4] R. Kumar, V. K. Garg, and S. I. Marcus, "On controllability and normality of discrete event dynamic systems," *Systems Control Lett.*, vol. 17, no. 3, pp. 157–168, 1991.

[5] R. Sreenivas, "A note on deciding the controllability of a language k with respect to a language l," *IEEE Trans. Autom. Control*, vol. 38, no. 4, pp. 658–662, April 1993.

[6] B. A. Brandin and W. M. Wonham, "Supervisory control of timed discrete-event systems," *IEEE Trans. Autom. Control*, vol. 39, no. 2, pp. 329–342, Feb. 1994.

[7] K. C. Wong and W. M. Wonham, "Hierarchical control of discrete-event systems," *Discrete Event Dynamic Systems*, vol. 6, pp. 241–273, 1996.

[8] R. Sengupta and S. Lafortune, "An optimal control theory for discrete event systems," *SIAM J. Control Optim.*, vol. 36, no. 2, pp. 488–541, 1998.

[9] H. Marchand, O. Boivineau, and S. LaFortune, "On the synthesis of optimal schedulers in discrete event control problems with multiple goals," *SIAM J. Control Optim.*, vol. 39, no. 2, pp. 512–532, 2000.

[10] A. Ray and S. Phoha, "A language measure for discrete event automata," in *International Federation of Automatic Control World Congress*, Barcelona, Spain, 2002.

[11] Y. Cao and M. Ying, "Supervisory Control of Fuzzy Discrete Event Systems," *IEEE Trans. Autom. Control*, vol. 35, no. 2, pp. 366–371, 2005.

[12] C. Griffin, "A note on deciding controllability in pushdown systems," *IEEE Trans. Autom. Control*, vol. 51, no. 2, pp. 334–337, February 2006.

[13] A. E. C. da Cunha and J. E. R. Cury, "Hierarchical Supervisory Control Based on Discrete Event Systems with Flexible Marking," *IEEE Trans. Autom. Control*, vol. 32, no. 12, pp. 2242–2253, 2007.

[14] C. Griffin, "A note on the properties of the supremal controllable sublanguage in pushdown systems," *IEEE Trans. Autom. Control*, In Press, 2008.

[15] C. G. Cassandras and S. LaFortune, *Introduction to Discrete Event Systems*. Boston, MA, USA: Kluwer Academic Publishers, 1999.

[16] J. Hopcroft and J. Ullman, *Automata, Languages and Computation*, 2nd ed. Upper-Saddle, NJ: Prentice-Hall Publisher, 1979.

[17] C. Griffin, "Decidability and Optimality in Pushdown Control Systems: A New Approach to Discrete Event Control," Ph.D. dissertation, Department of Industrial Engineering, The Pennsylvania State University, University Park, PA, December 2007.