# Extended Statecharts: A Specification Formalism for High Level Design

**A. Suraj, S. Ramaswamy, K.S. Barber**

The Laboratory for Intelligent Processes and Systems
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
http://www-lips.ece.utexas.edu
barber@mail.utexas.edu
(512) 471-6152

**Abstract**

This paper introduces Extended Statecharts as a comprehensive modeling mechanism for high level systems design. Extended Statecharts allow for the explicit representation of declarable problem-specific system soft failures, thereby allowing for failure related information to be incorporated into the high level system design. Temporal logic is used for verification of important design properties. An example of an assembly process is used to illustrate the capabilities of Extended Statecharts.

**Accepted to**

The Fourth International Conference on Control, Automation, Robotics and Vision
ICARCV'96, 3-6 December 1996
Westin Stamford, Singapore

# Extended Statecharts: A Specification Formalism for High Level Design *

**A. Suraj, S. Ramaswamy** ** **and K.S. Barber**

*Laboratory for Intelligent Processes and Systems, The Department of Electrical and Computer Engieering, The University of Texas at Austin, Austin, TX 78712-1084*

*Abstract* - **This paper introduces Extended Statecharts as a comprehensive modeling mechanism for high level systems design. Extended Statecharts allow for the explicit representation of declarable problem-specific system soft failures, thereby allowing for failure related information to be incorporated into the high level system design. Temporal logic is used for verification of important design properties. An example of an assembly process is used to illustrate the capabilities of Extended Statecharts.**

## 1. INTRODUCTION

Several software modeling tools and methodologies have been developed for system specification and analysis. The reader is referred to [1] for a detailed overview of such tools and methodologies. Extended Statecharts (ESC) are dynamic models of system behavior. The ESC representation provides a way to explicitly integrate failure information in the system design process. It allows data values to be associated with events.

The paper is organized as follows: Extended Statecharts, that further the use of Statecharts for high-level system design representation is presented in Section II. Section III illustrates the use of the ESC based approachin a simple assembly process. The final section, Section IV, concludes the paper.

## 2. EXTENDED STATECHARTS

In this section, Extended Statecharts [1], an extension to Statecharts both with respect to functional and temporal aspects is introduced. ESC is a 4-tuple (S, E, G, T), where, S is a set of states, E is a set of events, G is a set of guards and T is a set of transitions. It is assumed that the reader is familiar with statechart notations [3].

### 2.1 Basic ESC Notations

The set of states S is defined as, $S = \bigcup_{i=1}^{n} S^i$ where, $n$ is the number of levels in the model hierarchy. Exit-safe states represent states wherein a substate is in a stable state to process the transition out of a parent state [2]. That is, events causing a transition between higher level (parent) states are handled only if the substate of the parents' state is exit-safe. This extension explicitly allows the developer to specify certain non-interruptable critical operations. Non exit-safe states do not allow for higher level state transitions.

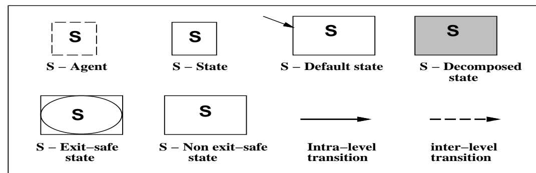


**Figure 1: ESC graphical notations**

Figure 1 illustrates the graphical notations associated with ESC-based designs. The dashed rectangle represents the definition of an *agent*. The solid rectangle is used to represent a state that is decomposed immediately. The arrow pointing to a state denotes a default state that the system reaches first among other states at the same level. The shaded rectangle is used to represent decomposed states. This means that the state S is decomposed further in another diagram. Intra-level and inter-level transitions are discussed in the next section.

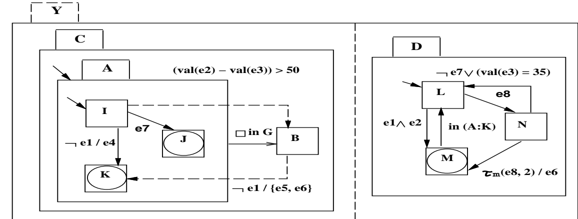


**Figure 2: ESC example graphical representation**

Figure 2 shows an example ESC representation. In the figure, **Y** is an *agent* that has states **C** and **D** in an AND composition. State **C** has substates **A** and **B** that are in a XOR combination, **A** being the default state. **D** has substates **L**, **M** and **N** where **L** is the default state. Notice that **J**, **K** and **M** are exit-safe states. Therefore, a transition from **Y** to another higher level parent state may occur if and only if the substate of **C** is either **J** or **K** and if substate of **D** is **M**.

### 2.2. Hierarchical Transfer of State Information

Two types of transitions are distinguished to manage the hierarchical information transfer between states. These include: *(i) Intra-level transitions* ($T_{intra}$): These are transitions between states in the same hierarchical level and *(ii) Inter-level Transitions* ($T_{inter}$): These are transitions between states in different hierarchical levels. Intra-level transitions are used to represent normal behavior and inter-level transitions are used to represent abnormal behavior. The graphical representation of these two transitions between states is shown in Figure 1. $T$, the set of all transitions is defined as $T \subseteq S \times L \times S$ where L is a set of transition labels. T is also defined as $T = T_{intra} \cup T_{inter}$. The distinction between these two transition types is essential in failure modeling.

There are two important rules that apply for inter-level transitions between states at two different hierarchical levels: *(i)* Transitions from a state at a lower hierarchical level to a state at a higher level are possible *only* when the lower level state is non exit-safe. *(ii)* Transitions from a state at a higher hierarchical level to a state at a lower level do not have such a restriction. A transition from a lower level state decomposition to a higher level state can occur on two conditions: *(i)* the operation is successful and *(ii)* the operation is unsuccessful and the system in that state requires external intervention (i.e. help) in proceeding further. The first phenomenon is implicitly captured by the notion of exit-safe states. The second phenomenon needs to be captured explicitly because it is a failure problem. Thus, we have inter-level transitions that occur only if the state at the lower level is non exit-safe.

---

* This research is supported in part by the Texas Higher Education Coordinating Board under grants ATP-452, ATPR-115 and ATPD-112.

** Dr. Ramaswamy is with the School of Computer and Applied Sciences, Georgia Southwestern State University, Americus, GA 31709. Between Aug. 94 and June 95 and subsequently in Summer 1996, Dr. Ramaswamy has been a visiting research fellow at the Laboratory for Intelligent Processes and Systems at the University of Texas at Austin.

## 2.3. Transition Labels

L is the set of transition labels defined as $L=\{(g,a)/g\in G, a\in A\subseteq 2^E\}$ where, G is the set of guards and the A is the set that contains subsets of all the events in the system.

Transition labels in ESC representation (see Figure 3) have a guard that acts as the input condition which must be satisfied for the transition to occur. This is followed by the event $\xi$, that always occurs (this is implicit in the transition labels and is not denoted in the examples). The "slash" (i.e. "/") is used to separate the input and the output processes. Output condition(s) or an action(s) can be a simple event or a set of events. This action event(s) is optional. If an action event(s) is specified, it is immediately generated and is regarded as an internal event that may cause transitions in other components. The input conditions and the event causing the transition are incorporated in the guard itself. This removes the need for having separate queues for events and input conditions during implementation.

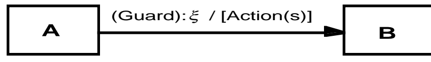


**Figure 3: Transition labels in ESC**

In Figure 2, the transition from **I** to **J** is caused by a guard *check(e7)* (i.e. e7). *check(e7)* is represented as just e7 for simplicity. This guard does not produce any action. But, the transition from state **I** to state **K** is caused by a guard ¬e1 followed by an action e4. This means event e4 is generated immediately. The transition from state **B** to state **K** produces two events e5 and e6 which are represented in the set {e5, e6}. Guards and the evaluation of guards to make transitions are discussed in the next section.

## 2.4. Guards on Transitions

E, the set of all events in the system is defined as $E=E_{BE}\cup E_{CE}$ where, $E_{BE}$ is the set of basic events and $E_{CE}$ is the set of composite events and $E_{BE}\cap E_{CE}=\phi$.

Guards are formed by the combination of four basic functions and different kinds of operators acting on events/states. These operators and functions may be combined to form complex conditions. Henceforth, complex conditions will be called as "complex events". The *functions* include:

(1) $in(x)$ which denotes that the system is in state x. $in(x{:}y)$ denotes that the system is in state y where, y is the immediate substate of x.

(2) $\tau_m(e,n)$ which denotes a time-out event that is generated if event $e$ happened $n$ steps earlier ($e\in E$ and $n\in N-\{0\}$, where N is the set of Natural numbers).

(3) $check(e)$ which returns true if event $e$ occurs and false if it does not occur. $check(e)$ is denoted as just $e$ for simplicity throughout this paper. In Figure 2, the transition from state I to K is caused by $\neg check(e1)/e4$ which is represented as $\neg e1/e4$.

(4) $val(e)$ which returns a data value associated with event $e$.

The *operators* used in the construction of guards are classified into three categories:

(1) *Arithmetic Operators* $(+,-,<,>,\leq,\geq,=,\neq)$: These operators act only on events that have associated data values. For example, consider two events $e_1$ and $e_2$ that represent temperature values ($e_1,e_2\in E$); then, the guard $g\in G$ may be $g{:}\ (val(e_1)+val(e_2))>20$. This means that the guard evaluates to true when the total value of the temperatures (i.e. $val(e_1)+val(e_2)$) becomes greater than $20$. Similarly, conditions such as 'equal' or 'less than' can be checked.

(2) *Logical Operators* ($\wedge$, $\vee$, $\neg$): The logical operators act on both events and states. If $g\in G$ and $g=e_1\vee e_2$ (or $e_1\wedge e_2$) where $e_1,e_2\in E$, $g$ becomes true when either event $e_1$ or $e_2$ (or $e_1$ and $e_2$) occur(s). In the same manner, $g=\neg e_1$ implies that the guard is true when $e_1$ does not occur. Some of the rules for the logical operators are summarized as follows: (a) $g\in G\Rightarrow\neg g\in G$ (b) $e\in E\Rightarrow e,\neg e\in G$ (c) $\tau_m(e,n)\in G$ and $\neg\tau_m(e,n)\in G$ (d) if $e_1,e_2\in G$ then $e_1\vee e_2, e_1\wedge e_2\in G$.

(3) *Temporal Operators* ($\circ$, $\Diamond$, $\mathbf{O}$, $U$): Within guard functions, the temporal operators act only on states. $\circ$ stands for always , $\Diamond$ stands for eventually , $\mathbf{O}$ stands for next and $U$ stands for until. If $s_1\in S$, g:$\circ$ $s_1$ (or g:$\circ in$ ( $s_1$)) is true if and only if *in* ( $s_1$) is true at the present step and continues to be true in all the steps from now on. g: $\Diamond$ $s_1$ (or g: $\Diamond in$ ( $s_1$)) is true if and only if *in* ( $s_1$) is true at least once in any step in the future. g: $\mathbf{O}$ $s_1$ (or g: $\mathbf{O} in$ ( $s_1$)) is true if and only if *in* ( $s_1$) is true in the next step (i.e. if the system reaches state $s_1$ in the next step). Finally, a guard g: $s_1\ U\ s_2$ (or g:*in* ( $s_1$) $U$ *in* ( $s_2$)) is true if and only if *in* ( $s_1$) is true until *in* ( $s_2$) (i.e. the system continues to be in state $s_1$ till $s_2$ is reached). The guard becomes false once $s_2$ is reached.

## 2.5. Failure Modeling

Potential system failures/errors, identified through sensory inputs, are classified into *soft* or *hard* failures. Error classification of previously known errors (and thus incorporated within the ESC design) occurs during the system modeling stage. So, when a system soft failure occurs, it might recover from the error and resume normal operations. System hard failures correspond to fatal failures from which the system cannot recover by normal mechanisms and needs external intervention. The system reaches a stop state when one of these errors occur. Errors due to hard failures are not explicitly modeled.

ESC designs integrate failure operations by exploiting the advantages of using XOR configurations with exit-safe states and the transition extensions proposed earlier in this section.

For example, in Figure 2, the higher level states **A** and **B** are in an XOR configuration where **A** represents the normal system operation and **B** represents the state associated with failure operations. In such a representation, the intra-level transition from state **A** to state **B** (caused by the guard $\circ in$ G) is never considered possible because it is assumed that there does not occur a normal transition (intra-level transition) from a normal state to an error state. However, the intra-level transition from state **A** to **B** in Figure 2 is used to illustrate the notion of exit-safe states.

# 3. EXAMPLE OF AN ASSEMBLY SYSTEM

In this section the modeling extensions discussed in the previous sections are illustrated by means of an example. The example illustrates the modeling concepts for representing an abstract high-level design. The block diagram in Figure 4 illustrates a factory floor consisting of a conveyor, a robot, two cutting machines, and an output buffer for the cutting machines.

A brief high level description of the system operations is as follows: Parts arriving on a conveyor are picked up by the robot and moved to one of the two cutting machines. As soon as the part arrives on the conveyor and is ready to be picked up (i.e. it enters within the robots' workspace), a sensor signals the robot. The robot checks for a free cutting machine; picks up the

part; and moves it to the appropriate cutting machine. Each part description is assumed to contain information related to the kind of job to be performed on it. The cutting machine uses this job description and performs the specific job and transfers the part on an output buffer. At some appropriate point, the robot moves the finished part from the buffer back to the conveyor. We are assuming that the parts arriving on the conveyor are well spaced in time which removes the need for an input buffer. The process of moving the part from the conveyor to either of the two cutting machines is the “in-path” and the process of moving it from the buffer to the conveyor is the “out-path” (Figure 4).

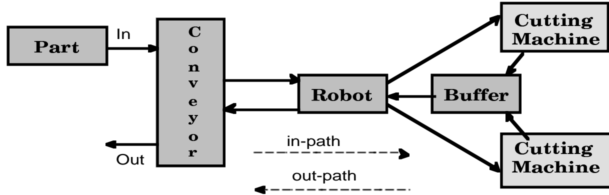


**Figure 4: Block diagram of assembly**

From the given problem description the following system agents are derived: *(i) Part:* The part agents carry information related to a particular part and the exact details of the job to be performed on the part. Additionally, the part agent also contains information about the part itself, such as its dimensions, type, etc. *(ii) Robot:* The robot agent contains information about the robot(s), its operations, availability, etc. *(iii) Cutting Machine:* The cutting machine agent contains information about the cutting process, the cutter(s) and their availability, *(iv) Buffer:* The buffer agent maintains the status of the output buffer that is used by the robot and cutting agents.

The *agents* 'PART', 'ROBOT', 'CUTTER1', 'CUTTER2' and 'BUFFER' are in an AND composition (Figure 5). Tables I and II provide a complete description of all the events associated with the state transitions. In the rest of this section, ESC models of the above agents are described.

| | | | |
|---|---|---|---|
| **e0** | SensorSignal | **e1** | Pickupfromconveyor |
| **e2** | Pickupfrombuffer | **e3** | RobotMalfunction |
| **e4** | PickupInitComplete | **e5** | C1Free |
| **e6** | C2Free | **e7** | RobotDroppedPart |
| **e8** | PickupComplete | **e9** | MoveInitComplete |
| **e10a** | Cutter1Reached | **e10b** | Cutter2Reached |
| **e11** | ConveyorReached | **e12** | CutterInitComplete |
| **e13** | Move Complete | **e14** | Moving Done |
| **e15** | RobotArmRelease | **e16** | JobDescription |
| **e17a** | Cutter1Malfunction | **e17b** | Cutter2Malfunction |
| **e18** | CutFinished | **e19** | CutProcessComplete |
| **e20** | BufferCountDecremented | **e21** | BufferCountIncremented |
| **e22** | BufferFull | **e23** | BufferEmpty |
| **e24** | PartReachable | **e25** | ResetRobot |
| **e26** | RobotHardFailure | **e27a** | ResetCutter1 |
| **e27b** | ResetCutter2 | **e28a** | Cutter1HardFailure |
| **e28b** | Cutter2HardFailure | | |

**Table I: Basic Events**

| | |
|---|---|
| **c1** | $e1 \vee e2$ |
| **c2** | $e4 \wedge ((e5 \vee e6) \vee e21)$ |
| **c3** | $(\neg e3 \wedge ((e10a \vee e10b) \vee e11)) / \{e13, e14\}$ |
| **c4** | $(\neg e3 \wedge \neg ((e10a \vee e10b) \vee e11)) / e14$ |
| **c5** | $(e24 \vee e25 \vee e27a \vee e27b) \wedge \neg (e26 \wedge e28a \wedge e28b)$ |
| **c6** | $\neg$ e0 $\wedge$ (*in* (Buffer:full) U *in* (Buffer:Empty) ) / $\neg$ e1 |
| **c7** | *in* (Buffer:Empty) |
| **c8** | $(e15 \wedge e11) / e20$ |
| **c9** | $e15 \wedge (e10a \vee e10b)$ |
| **c10** | *in* (Cutterx: Put on Buffer) (x = a or b) |
| **c11** | $e17a \wedge e17b$ |
| **c12** | $e3 \vee e7$ |
| **c13** | *in* (move:move succeeded) / e15 |
| **c14** | $\neg$ e3 / e4 |
| **c15** | $\neg$ e7 / e8 |
| **c16** | $e3 \vee (e17a \wedge e17b)$ |
| **c17** | *in* (Pickup:Pickup Aborted) |
| **c18** | *in* (Move:Move failed) |
| **c19** | $\neg$ e3 / e9 |
| **c20** | $(e15 \wedge e10a) / \neg e5$ |
| **c21** | e2 / {e5, e21} |
| **c22** | (e19 $\wedge \neg$ *in* “Buffer:Full”) / e2 |
| **c23** | $\neg$ e17a / e12 |
| **c24** | $e12 \wedge e16$ |
| **c25** | $(\neg e17a \wedge e18) / e19$ |
| **c26** | *in* (Cut:Cut Failed) / $\neg$ e5 |
| **c27** | $\neg e24 \wedge e25 \wedge \neg e26$ |
| **c28** | $e27a \wedge \neg e28a$ |

**Table II: Composite Events**

## *3.1 Part Agent*

The part enters the system and defaults to the 'on conveyor' state (Figure 5). If the event e0 occurs (i.e. a signal for the robot to pickup part) it enters the exit-safe state 'ready for pickup'. If the event $\neg$e0 occurs and the buffer is full at the same time, the system is designed to process only the "out-path" (composite event c6) and transitions into the 'suspend' state. Once the buffer is empty, it returns from the 'suspend' state to the 'on conveyor' default state. At the higher-level, when the part is ready for pickup (the event 'e1') by the robot (Note that the system needs to be in the exit-safe state 'Ready for pickup' of the 'into conveyor' state for it to process event e1 and go to the 'robot move' state), the part enters the 'robot move' state from the 'into conveyor' state. In the 'robot move' state, the part is picked up, moved and then putdown on one of the cutters. This results in the part moving into the 'into cutter' state after the robot reaches the cutter, placing the part on the cutter and releasing its arm (this transition is caused by the composite event c9). Once the part is cut, it enters the 'into buffer' state when the system is in the 'put on buffer' state of the 'CUTTER1' or 'CUTTER2' (see event c10 in Table II). The event e2 triggers the transition from 'into buffer' to the 'robot move' state. Once the conveyor is reached and the robot arm is released (event $(e15 \wedge e11)$ ), the part is 'back to conveyor' and produces event e20 (see event c8 in Table II) which decrements the buffer count.

The Part Agent transitions into the 'Error' state if both the cutters malfunction (event c11) or when the robot drops a part or malfunctions (event c12) which can occur when the PART is in states 'into cutter' or 'robot move' states respectively. The system recovers from the error state if a soft failure occurs in the system i.e. if the part is reachable, the robot is reset or the cutters are reset.

## *3.2 Robot Agent*

The 'ROBOT' (Figure 5) is in the 'idle' state by default and transitions into the 'pickup' state when the part is ready for pickup from the conveyor or the buffer (event $c1 = e1 \vee e2$). The robot starts the initialization process and enters the 'init complete' exit-safe state if there is no malfunction (event c14) or it enters the 'pickup aborted' state if there is a malfunction (event c16). It starts the pickup process once the initialization is complete and when either of the two cutters are free (event c2). If the robot does not drop the part, it transitions into the 'pickup succeeded' state and if it does drop the part while in motion, it enters the 'pickup aborted' state (event e7). Notice that the pickup is aborted if either the

robot malfunctions, robot drops part or when both the cutters malfunction. This could happen during initialization or during the pickup process itself. The system enters the 'move' state once the pickup is complete (event e8).

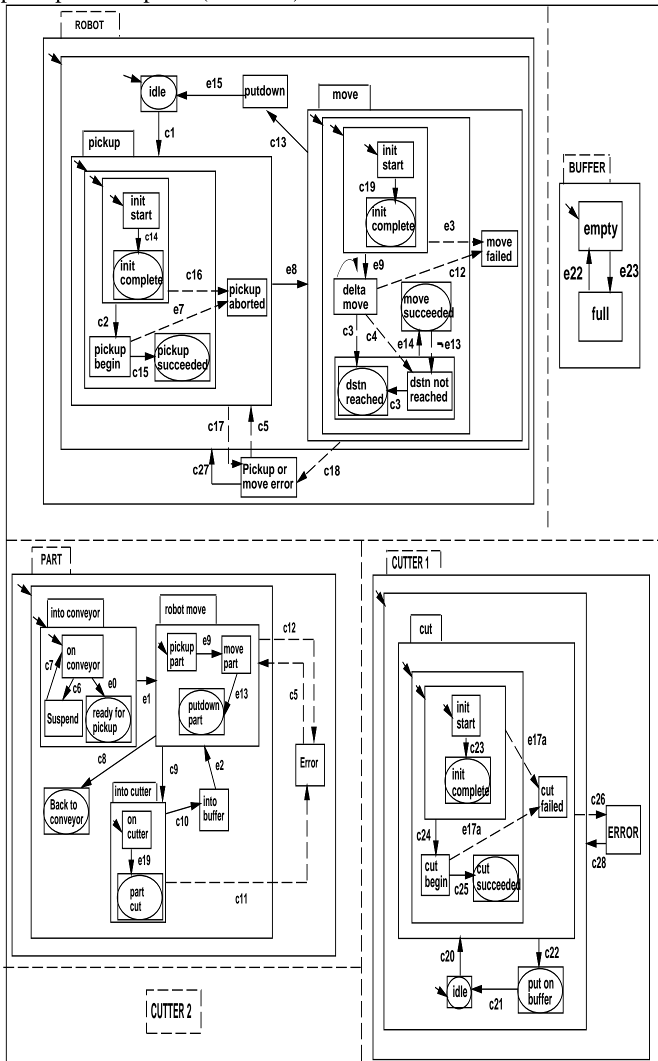


**Figure 5: ESC model of the Assembly System**

In the 'move' state, the robot starts the initialization for the move process and enters the ‘init complete’ exit-safe state. When the initialization is successful, the robot moves to the 'delta move' state. If the initialization is unsuccessful, the robot transitions into the 'move failed' state. From the 'delta move' state the robot transitions to ‘dstn reached’ if the destination is reached (i.e. either of the cutters or the conveyor is reached) or into ‘dstn not reached’ if its not. Note that both the events c3 and c4 produce the action event e14 which causes a transition to ‘move succeeded’ and only c3 produces both e13 and e14. In the ‘move succeeded’ state, if ¬e13 occurs, it takes the system back to the ‘dstn not reached’ state. Therefore the robot makes certain moves until the destination is reached.

In the 'move' state, robot malfunction can occur during initialization or the move process. The move fails when there is a malfunction or if the robot drops a part. Once the exit-safe state ‘move succeeded’ is reached, the robot agent transitions from ‘move’ to the ‘putdown’ state (event c13) in ROBOT. In this state, the robot places the part on the conveyor or the cutter depending on whether the robot processing the “in-path” or the “out-path” (see Figure 4). The robot transitions back to the ‘idle’ state when the robot arm is released.

In the ROBOT agent, errors can occur during ‘pickup’ or ‘move’ (events c17 and c18) and the system transitions into ‘Pickup or Move error’. If the error occurred due to the dropping of the part by the robot and if the part is reachable (event c5), the system recovers from the error by transitioning into the ‘pickup’ state again. If both the cutters fail, the robot waits for either of the cutters to recover and then transitions back into ‘pickup’ state (event c5). If the error occurred due to a robot malfunction and if it is not a hard failure, the robot is reset and returns to the default ‘idle’ state (event c27).

### *3.3 Cutter Agent*

The 'CUTTER1' (Figure 5) is 'idle' by default and transitions into the 'cut' state once cutter1 is reached and the robot's arm is released (event c20) then produces event ¬e20 signifying that cutter1 is not free anymore. Cutter1 starts the initialization process in the ‘cut’ state and transitions into the ‘init complete’ exit-safe state if there is no malfunction. Once the initialization is complete and when it gets the job description from the part (event c24), cutter1 starts the cutting process in the state ‘cut begin’. Cutter1 transitions to the ‘cut failed’ state if there is a cutter1 malfunction (event e17a). Once the cutting process is complete, it transitions to the ‘cut succeeded’ state (event c25) from the 'cut begin' state. The ‘cut failed’ state is reached during initialization or during the cutting process.

Cutter1 transitions into the ‘put on buffer’ state when the cutting process is complete and if the buffer is not full and signals that the part is on the buffer by producing event e2 (event c22). Cutter1 transitions back to the ‘idle’ state and signals it is free and the buffer count is incremented (event c21). Cutter1 transitions into the ‘Error’ state from ‘cut’ only when cutter1 malfunctions (event c26). Cutter1 signals the system that it is not free anymore and transitions back to the ‘idle’ state if its reset as long as its not a hard failure (event c28).

The decomposition of the agent CUTTER2 is identical to the CUTTER1 and hence is not shown separately.

### *3.4 Buffer Agent*

The BUFFER agent (Figure 5) is either 'empty' by default or 'full'. The events e22 and e23 signify the transitions between these states.

## 4. ESC DESIGN VERIFICATION

The most important properties with respect to manufacturing control software are: *liveness* and *safety*. The *liveness* property verifies that the system keeps functioning under certain conditions. The *safety* property assures that certain error events will not occur and thereby ensures that the system functions normally. Formal analysis of an ESC design includes proofs of both the *liveness* and *safety* properties. The proofs presented in the following sections will use the properties of propositional and temporal logic listed below [1]. Sections 4.1 and 4.2 present two distinct liveness and safety properties for the system.[1]

If $w$ , $w_1$ and $w_2$ are logical formulas,

**P1**: $\neg\Diamond w \equiv \circ\ \neg w$, **P2:** $\Diamond\neg w \equiv \neg \circ\ w$,
**P3**: $\neg(w_1 \wedge w_2) \equiv (\neg w_1 \vee \neg w_2)$, **P4:** $\neg(w_1 \vee w_2) \equiv (\neg w_1 \wedge \neg w_2)$,
**P5**: $(w_1 \Rightarrow w_2) \equiv (\neg w_1 \vee w_2)$, **P6**: $\Diamond(w_1 \wedge w_2) \equiv (\Diamond w_1 \wedge \Diamond w_2)$,

[1] Although other liveness and safety properties for the example may be defined, we are interested in capturing the important characteristics of the high level design representation and the following safety and liveness properties are sufficient to address this concern. Therefore, we describe only two liveness and two safety properties which are most relevant to the desirable characteristics of the high level design specification.

**P7**: $\Diamond(w_1 \vee w_2) \equiv (\Diamond w_1 \vee \Diamond w_2)$,
**P8**: $w_1 \wedge (w_2 \vee w_3) \equiv (w_1 \wedge w_3) \vee (w_1 \wedge w_3)$.

### 4.1 Liveness Properties

The liveness properties specify the events that can occur and the states that can be reached in a system.

**Liveness Property 1:**

When the Sensor Signal occurs (i.e. the part is ready for pickup from the conveyor) and the robot is operational, the part is moved to one of the cutters eventually. The proof of this property is given below.

(1) $\circ[(e0 \wedge \neg e3) \Rightarrow \Diamond$ in ("Part:into cutter")$]$

(2) From ESC decomposition and assuming $e10 = e10a \vee e10b$, we get $\circ[(e0 \wedge \neg e3) \Rightarrow \Diamond (e15 \wedge e10)]$

(3) Assuming the negation of Liveness Property 1: $\neg\circ[(e0 \wedge \neg e3) \Rightarrow \Diamond (e15 \wedge e10)]$

(4) Applying P5: $\neg\circ[\neg(e0 \wedge \neg e3) \vee \Diamond (e15 \wedge e10)]$

(5) Applying P2: $\Diamond\neg[\neg(e0 \wedge \neg e3) \vee \Diamond (e15 \wedge e10)]$

(6) Applying P4 and P6: $\Diamond[(e0 \wedge \neg e3) \wedge \neg(\Diamond e15 \wedge \Diamond e10)]$

(7) Applying P3 : $\Diamond[(e0 \wedge \neg e3) \wedge (\neg\Diamond e15 \vee \neg\Diamond e10)]$

(8) Applying P8:
$\Diamond[(e0 \wedge \neg e3 \wedge \neg\Diamond e15) \vee (e0 \wedge \neg e3 \wedge \neg\Diamond e10)]$

(9) Applying P7 and P1: $\Diamond(e0 \wedge \neg e3 \wedge \circ\neg e15) \vee \Diamond(e0 \wedge \neg e3 \wedge \circ\neg e10)$

From the conclusion of step 9, the first disjunct signifies that eventually, e0 , ¬e3 and permanently ¬e15 occur. Consider the composite event c13 = e14 / e15, ¬e15 implies that e14 does not occur which further implies that 'move succeeded' state is never reached inside the 'move state'. This can happen only when the robot malfunctions which produces event e3. The first disjunct produces a contradiction since ¬e3 occurs.

The second disjunct can be broken down into $\Diamond(e0 \wedge \neg e3 \wedge \circ(\neg e10a \wedge \neg e10b))$ by applying P4. The above disjunct causes event $c4 = (\neg e3 \wedge \neg((e10a \vee e10b) \vee e11)) / e14$ (since its guard is true) which takes the event to the state "dstn not reached" (Assuming that the system is in the "delta move" state at the present step). The system can never get to the "dstn reached" state since $c3 = (\neg e3 \wedge ((e10a \vee e10b) \vee e11)) / \{ e13, e14 \}$ can never occur. This would cause event e3 to occur which is a contradiction.

The negation of Liveness Property 1 produces a contradiction which proves the property.

**Liveness Property 2:**

If the part is cut successfully and the robot is operational, it eventually ends up back on the conveyor. That is,

(1) $\circ[$(*in* ("cut:cut succeeded") $\wedge \neg e3) \Rightarrow \Diamond$ *in* ("Part:back to conveyor")$]$

(2) From ESC decomposition : $\circ[(\neg e17a \wedge e18 \wedge \neg e3) \Rightarrow \Diamond(e15 \wedge e11)]$

Following steps 3 to 9 just as in proof of liveness property 1, we get $\Diamond(\neg e17a \wedge e18 \wedge \neg e3 \wedge \circ\neg e15) \vee \Diamond(\neg e17a \wedge e18 \wedge \neg e3 \wedge \circ\neg e11)$

From the proof of Liveness property 1, we can see that ○¬e15 implies that e3 occurs which is a contradiction of the first disjunct. Similarly for the second disjunct, we can see that ○¬e11 implies e3 would occur which is a contradiction. Hence both the disjuncts produce a contradiction of the negation of Liveness Property 2 and this proves the property.

### 4.2 Safety Properties

Safety properties specify the events or states that may not occur in a system.

Whenever a soft failure occurs, the system eventually gets back to normal operation. For a robot, the soft failure can occur (under the assumptions that a robot hard failure does not occur in the system) if the (1) robot dropped the part or if (2) robot malfunctions.

**Safety Property 1:**

If the robot drops a part and the part is reachable and if its not a hard failure, the robot picks up the part eventually. That is,

(1) $\circ[(e7 \wedge e24 \wedge \neg e26) \Rightarrow \Diamond$ *in* ("Robot:pickup")$]$

(2) From ESC decomposition : $\circ[(e7 \wedge e24 \wedge \neg e26) \Rightarrow \Diamond c5]$

(3) Rewriting $c5 = (a \wedge \neg b)$ where $a = (e24 \vee e25 \vee e27 \vee e27b)$ and $b = (e26 \wedge e28a \wedge e28b)$ :
$\circ[(e7 \wedge e24 \wedge \neg e26) \Rightarrow \Diamond (a \wedge \neg b)]$

(4) Assuming the negation of Safety Property 1:
$\neg\circ[(e7 \wedge e24 \wedge \neg e26) \Rightarrow \Diamond (a \wedge \neg b)]$

(5) Applying properties P5, P2, P4, P6, P3, P8, P7 and P1: $\Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ\neg a) \vee \Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ b)$

(6) Substituting for a and b:
$\Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ\neg(e24 \vee e25 \vee e27a \vee e27b)) \vee \Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ(e26 \wedge e28a \wedge e28b))$

(7) Applying P4:
$\Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ(\neg e24 \wedge \neg e25 \wedge \neg e7a \wedge \neg e27b)) \vee \Diamond(e7 \wedge e24 \wedge \neg e26 \wedge \circ(e26 \wedge e28a \wedge e28b))$

In step 7, we can see the direct contradiction between the events e24 and ¬e24 in the first disjunct and between the events e26 and ¬e26 in the second.

**Safety Property 2:**

If the robot malfunctions and its not a hard failure, the robot resets itself eventually. That is,

(1) $\circ[(e3 \wedge \neg e26) \Rightarrow \Diamond e25]$

(2) Negation of Safety Property 2: $\neg\circ[(e3 \wedge \neg e26) \Rightarrow \Diamond e25]$

(3) Applying properties P5, P2, P4 and P1: $\Diamond[e3 \wedge \neg e26 \wedge \circ\neg e25]$

The final statement states that eventually, events e3, ¬e26 and permanently ¬e25 will occur. The contradiction comes from the fact that permanently ¬e25 can occur only when there is a hard failure in the system i.e. if event e26 occurs and the occurrence of ¬e26 contradicts this fact. This proves Safety Property 2.

## 4. CONCLUSIONS

In this paper, ESC have been introduced. The notion of exit-safe states and event/transition extensions in ESC are shown to be useful for the high level design of reactive systems. The different functions and operators defined on the events were shown to provide a complete event structure.

## References


1. Suraj, A., *Extended Statecharts for Systems Modeling, Specification and Verification*, August 1996, Master's thesis, Dept.. of Electrical and Computer Engg., The University of Texas at Austin.
2. Barcio, B., *SMOOCHES: State Machines for Object-Oriented Concurrent, Hierarchical Engineering Specifications*,, December 1994, Master's thesis, Dept. of Electrical and Computer Engg., The University of Texas at Austin.
3. Harel, D. *Statecharts: A Visual Formalism for Complex Systems*. in *Science of Computer Programming*. June 1987. 8(3): p 231-274.