Intrinsic Evolution of Safe Control Strategies for Autonomous Spacecraft

Garrison W. Greenwood, Senior Member, IEEE[*]
Dept. of Electrical & Computer Engineering
Portland State University, Portland, OR 97207

Abstract

Autonomous space vehicles need adaptive control strategies that can accommodate unanticipated environmental conditions. Although it is not difficult to construct alternative control strategies, a proper evaluation frequently can only be done by actually trying them out in the real physical environment. It therefore becomes imperative that any candidate control strategy be deemed safe—i.e., it won't damage any systems—prior to being tested online. How to do this has been a challenging problem.

We propose a solution to this problem. Our approach uses an evolutionary algorithm to intrinsically evolve new control strategies. All candidate strategies will be checked for safety using formal methods. More specifically, an evolutionary algorithm will evolve a series of finite state machines, each of which encodes a unique control strategy. Model checking will guarantee whether all safety properties are satisfied in the strategy. A numerical example is included to illustrate our approach.

Key Words: evolutionary algorithm, control strategy, formal methods

[*]greenwood@ieee.org

1 Introduction

In October 1997 NASA launched the Cassini spacecraft to explore Saturn and its moons. Due to arrive at Saturn in 2004, the spacecraft will eventually establish an orbit around the moon Titan and then launch a probe, called Huygen, into the moon's atmosphere. Atmospheric data is relayed from Huygen to the Cassini spacecraft and then transmitted back to earth. Engineers discovered last year that communications between the Huygen probe and the Cassini spacecraft would fail due to unanticipated Doppler effects, which could lead to a total loss of data on Titan's atmosphere. Fortunately, this problem was discovered in time and flight plan changes are being implemented to compensate for the Doppler effect [1].

But there is an even more important issue to consider: adaptive control strategies for autonomous spacecraft. Control strategies are critical ingredients of a space mission because they indicate what actions are to be taken by the spacecraft in response to environmental conditions. The Doppler problem on the Cassini spacecraft would only cause loss of data. Although such a loss is not to be taken lightly, it is inconsequential when compared to the loss of the spacecraft itself, which is entirely possible if the system is trying to adapt to an unanticipated environmental condition by switching to a new—and presently undefined—control strategy.

Reconfiguration was able to correct the Cassini spacecraft problem, and it may well prove to be the key to handling a whole host of such problems. More specifically, *reconfigurable circuitry* which can adopt different functionality can help to compensate for unanticipated environmental conditions. Reconfigurable circuitry or systems are also beneficial for fixing failures because it eliminates the need for redundant hardware—which consumes precious space and weight—by simply reconfiguring the existing hardware to compensate for the failure. But, despite the enormous advantages of reconfiguration, there remains a very important question:

Question 1 *If new reconfiguration information must originate from Earth, will it arrive in* *time to do any good?*

Communications between Earth and Mars takes around 10 minutes, which means the likelihood of receiving new configuration information for deep space missions—in a timely manner—is not good. Consequently, we must answer Question 1 in the negative.

The solution to the problem raised by Question 1 may lie with *adaptive systems*—i.e., systems capable of reconfiguring themselves in response to faults or a changing operational environment. Of particular interest is whether this adaption can be performed *in-situ* (in place), which removes any reliance upon Earth-bound resources for new configuration information. Stoica et. al [2] points out that much of the previous work on adaptive systems has been restricted to sensors and signal-conditioning circuitry because of the dire consequences of evolving an unsafe control strategy. Nevertheless, the ability to adaptively control spacecraft—without requiring human intervention—is still an area of enormous research interest [3].

We believe our approach is the first step towards solving this problem. Specifically, we have developed a method for adapting control strategies in ways that are guaranteed to be disaster-free during the reconfiguration process. This paper documents the key elements of our method.

2 Overview of our approach

Our basic approach is to evolve a series of deterministic *finite state machines* (FSMs), each which encodes a potentially new control strategy. The efficacy of each strategy will be assessed by actually trying it in the real physical environment. However, this will only be done with control strategies certified as being safe. The problem is formulated in such a way the *formal verification techniques* can guarantee whether or not the safety properties are met in the control strategy.

A FSM is a digraph that completely describes a control strategy. Each state is a stable system condition and transitions between states occur based on new input information. The outputs associated with each state are command signals issued to the system being controlled. A hallmark of FSMs is the processing of inputs depends on the current state of the system. In other words, the same input applied at two different times may not illicit the same output response because the system may have been in different states at those two time periods.

FSM design requires completing several tasks: (*i*) define the number of states, (*ii*) define the outputs in each state, and (*iii*) define the transitions conditions between states. In many instances a design engineer could hand-code the FSM, but this can be tedious if the control strategy is complex. The solution space of all FSMs that apply to the problem of interest is often extremely large. This means a complete enumeration and evaluation of all solutions is impractical. Even deterministic searches through the solution space may simply take too long. Thus, in practice, only a stochastic search algorithm is likely to be successful in identifying an acceptable FSM.

Researchers have found much of the FSM design effort can be alleviated by using *evolutionary algorithms* (EAs). These are an extremely powerful class of stochastic search algorithms that use the principles of Darwinian evolution found in Nature to conduct searches. More specifically, a population of solutions undergoes stochastic modification to create new candidate solutions. Each solution is assigned a fitness value that reflects the quality of the solution. A survival of the fittest criteria—i.e., those solutions with the highest fitness value—determines which solutions survive to reproduce in future generations. Done properly, the entire population evolves towards regions of the search space that contains optimal solutions. With respect to the problem of interest, each solution in the population is a FSM and its fitness measures the acceptability of its encoded control strategy—i.e., the better the control strategy performs, the higher its fitness value. New control strategies are created from existing strategies by any stochastic modification to a FSM. For instance, randomly adding or deleting a state, changing a state's output, or changing the position of an arc would create a new control strategy.

There are two ways of determining if an evolved control strategy is acceptable: an *extrinsic evolution* where the strategy is simulated first and only the one best strategy is actually implemented, or an *intrinsic evolution* where every candidate strategy is downloaded into the system and exercised in the real physical environment. Evolutionary algorithms can work with either type, but the extrinsic evolution may be problematic. EAs typically have some closed-form objective function that assigns fitness values. Unfortunately, it may not always be possible to define an appropriate objective function for a needed control strategy. This means the only way of analyzing a control strategy may be to actually try it out in the real physical environment. In other words, in many cases intrinsic evolution may be the only thing that makes sense. It is therefore absolutely essential that the control strategy be safe—i.e., it does no harm to the controller itself nor to any other system—while it is being tested for suitability. This presents what heretofore has been an open challenge:

Question 2 *Since an EA creates new FSMs randomly, is there some way to know if the encoded control strategy is safe before it is tested in the real physical environment?*

We believe our approach answers Question 2 in the affirmative. We will use an EA that searches for the optimal control strategy by creating candidate FSMs. However, only control strategies that pass a safety check will be downloaded for evaluation. We will borrow automatic formal verification methods to assess this safety. These methods use mathematically provable techniques to characterize a system without conducting exhaustive simulation or testing. Specifically, we will rely on *model checking* techniques [4] to verify the safety of candidate FSMs generated by the EA. Although model checking has been extensively used in hardware design and software verification, to the best of our knowledge no prior research effort in formal methods has attempted the problem we consider here. Figure 1 shows the control strategy development environment.

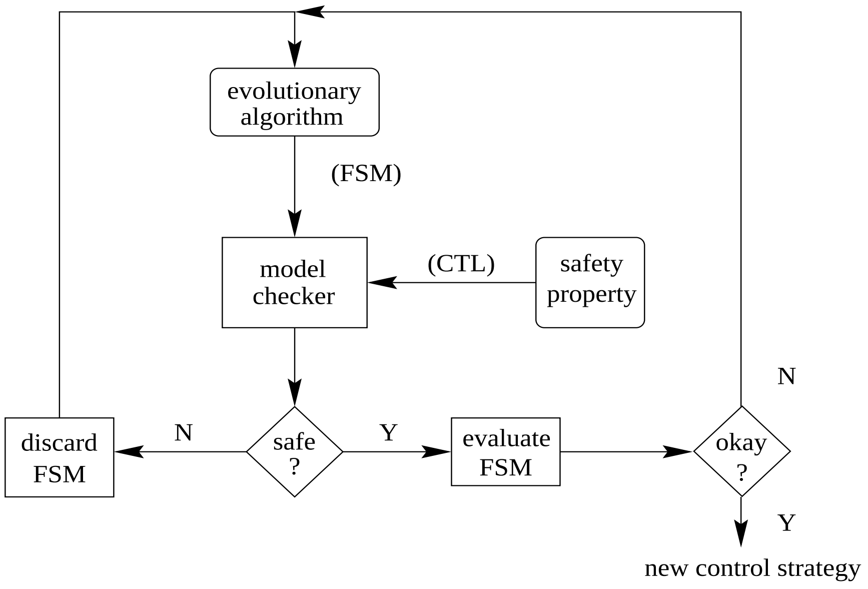

Figure 1: Conceptual diagram showing how to evolve safe control strategies. Each candidate control strategy evolved by the evolutionary algorithm is described by a finite state machine (FSM). All safety properties are stated as computational tree logic (CTL) expressions. The Model Checker uses formal verification techniques to determine if the evolved control strategy is safe. Safe control strategies are evaluated in the physical environment whereas unsafe strategies are immediately discarded and a replacement strategy is evolved. Once an acceptable new control strategy is found, it replaces the current control strategy.

3 Background

3.1 Finite State Machines

A (deterministic) finite state machine $\mathcal{M}$ is defined by a 6-tuple

$$\mathcal{M} \;=\; (I, O, S, \delta, \gamma, S^o)$$

where

I	is the set of inputs
O	is the set of outputs
S	is the set of states
$\delta : I \times S \rightarrow S$	is the state transition function
$\gamma : I \times S \rightarrow O$	is the output mapping function
S^o	is the initial state

For a FSM that encodes a control strategy, the states are fixed conditions of the system under control, the inputs are measurements of the physical environment, the outputs are commands to the system, and δ defines the next system states based on the current input and the current state. We will illustrate in Section 4 how a control strategy is completely described by a FSM. We will also show how its safety can be verified.

As a side comment, our method requires a data structure that satisfies two criteria: (1) it completely encodes all aspects of the control strategy, and (2) it is compatible with existing model checkers. We have chosen a FSM as the data structure. It is true that FSMs can be converted to logic circuits—but that does not mean we are solving a logic synthesis problem. We are not designing a digital circuit. We are trying to evolve a control strategy. In principle our method can use any data structure so long as it satisfies the above two criteria.

3.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of search, learning, and optimization methods based on analogies to Darwin's theory of natural selection. The *genetic algorithm*(GA) [5, 6] is the most widely known form of EA and they have been applied to a number of scientific and technical problems. Fogel, et al. [7] independently developed an evolutionary algorithm called *evolutionary programming* (EP) that evolves finite state machines to induce sequential patterns and make predictions. Still another EA called an *evolutionary strategy* (ES) was independently developed by Rechenberg [8]. All EAs share the same basic organization:

iterations of competitive selection and random variation. Unlike traditional methods, every EA processes a population of potential solutions in parallel rather than just a single solution. However, it has been shown that no one type of EA—or for that matter any other kind of non-EA algorithm—performs optimally over all problem classes [9].

The GA is arguably the most widely known EA, and it is almost exclusively used by the evolvable hardware community (e.g, see [10]). However, we have decided not to use a GA but to instead adopt a variant of the EP algorithm. The most compelling reason for not using GAs is they are not well suited for evolving structures—which is precisely what we want our EA to do! (See [11] for a discussion on generating structures with GAs.) Conversely, EP is ideally suited for such applications. In fact, its very first application was designing FSMs [7]. It is for these reasons we have decided to use an EA similar to an EP to evolve our FSM structure. Our EA differs from the pure EP in two ways: tournament selection is not used to rank the entire population, and adaptive mutation strengths are not used. Our EA is described in Figure 2.

```
1. randomly create an initial population of $\mu$ FSMs
2. evaluate all FSMs (see Section 4.2)
3. create $\mu$ new FSMs by repeating the following two steps:
     (i) conduct a binary tournament to select a parent FSM
         from the current population
     (ii) randomly mutate the parent FSM to create a new
          offspring FSM
4. evaluate the combined population of $\mu$ parent FSMs and
   $\mu$ offspring FSMs. rank them according to fitness.
5. save the $\mu$ best ranked FSMs and discard the others.
6. exit if termination criteria is met. otherwise, go to step 3.
```

Figure 2: EA for creating FSMs. Note that the algorithm as shown does not make any safety checks. See Section 4.3 for details on modifying the algorithm to add the necessary safety checks.

The initial population was randomly generated and its safety can be assured by a variety of offline methods—including exhaustive testing. But the offspring are randomly generated on-the-fly and extensive testing to verify safety takes too long to be practical. We will use *symbolic model checking* techniques to verify the control strategy safety. This is a well established formal verification technique that quickly verifies if a proposed control strategy satisfies the necessary safety properties. In the next section we will discuss the safety checking in depth. Specific details for implementing the EA are differed until Section 4.

3.3 Model Checking

Our goal is to check that the control strategy satisfies critical behavioral properties to ensure reliable and correct functioning. *Model checking* (MC) is an advanced formal method which has potential to help us achieve that goal [12].

There are three methods of functional verification: simulation, emulation, and formal verification [13]. Simulation and emulation are widely used, but they do have one inherent problem: they are good at proving the *presence* of unsafe conditions, but they are not so good at proving the *absence* of unsafe conditions.

Formal verification methods do not rely on running test inputs through a system to determine its behavior. Rather, these methods use mathematical techniques to examine the entire solution space for a specified design property [13]. There is no need to construct test vectors or patterns. Moreover, the results are guaranteed—i.e., if formal verification says a property is verified, then it exists under all conditions. What makes this possible is formal verification methods employ mathematical logic and can theoretically account for every possible situation.

MC is a formal method that verifies if a system, modelled as a FSM, adheres to a specified property. The properties of interest are encoded as temporal logic expressions. Temporal logic is just a formal way of expressing properties that change over time [14]. There are many different kinds of temporal logic but *computation tree logic* (CTL) is the most widely used with model checkers. The basic idea is that we start with ordinary Boolean logic, and then add special temporal operators for describing future events. For example, in CTL, the operator **AX** means "for all possible input observations, in the next state,...", the operator **EX** means "there exists an input such that in the next state,...", the operator **AG** means "for all possible input observations, it will always be true that,...", the operator **EF** means "there exists a sequence of input observations such that eventually...", and so forth. The temporal operators can nest, so for example, **AGEF**(reset) says that it is always possible to find a path back to reset, and **AG**(req) $\Rightarrow$ **AF**(ack) says that every request is always eventually followed by an acknowledgment. It is also possible to express properties using propositional connectives. For example, if f and g are CTL formulas, so are $\neg f$, $f \wedge g$, and $f \vee \neg g$.

The FSM states are labeled with the safety properties that hold in that state. We can now transform the FSM into a *Kripke structure* $M(S, I, R, L)$ where

S	is the set of states
I	is the set of initial states
$R \subseteq S \times S$	is the set of transitions
$L : S \rightarrow 2^{AP}$	is a labeling function
AP	is the set of atomic propositions (i.e., safety properties)

A path $\pi = s_0, s_1, s_2, \ldots$ through the control strategy (where s_0 is the initial state and $R(s_i, s_{i+1})$ holds for all i), describes the sequence of actions to be taken by a system in response to a sequence of observed inputs. The MC problem can then be described as follows:

Given a Kripke Structure $M(S, I, R, L)$ representing a control strategy, and a safety property f to be verified, find the set of states that satisfy

$$\{s \in S \mid M, s \models f\}$$

A graphical representation of the MC algorithm is shown in Figure 3. This algorithm can be executed in $O(|S| + |R|)$ time.

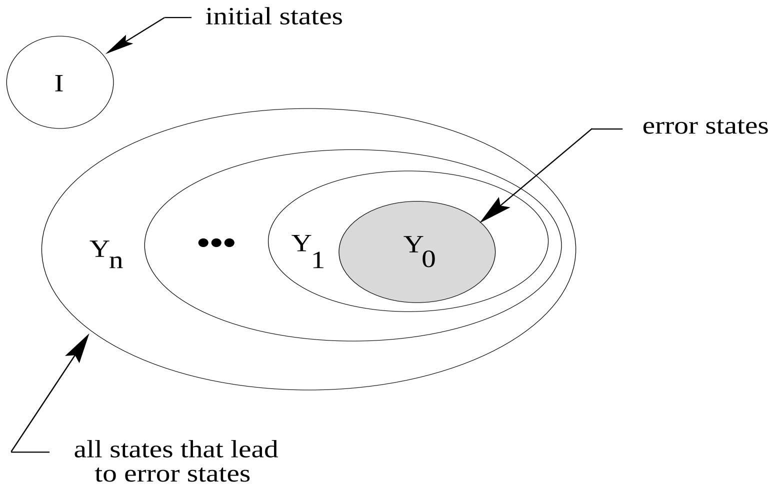

Figure 3: A graphical depiction of the model checking algorithm. I is the set of all initial states and Y_0 is the set of states that violates a safety property. The algorithm recursively computes $Y_{i+1} = \mathrm{Pre}(Y_i) \bigcup Y_i$ for $i = 0, 1, 2, \ldots n-1$ where $\mathrm{Pre}(Y_i)$ is the preimage of the set Y_i. Y_n then represents the set of all states that can reach an error state. The system is safe if $Y_n \bigcap I = \emptyset$. This check can be done in linear time.

We are concerned with sets of states rather than a single state. These sets can be represented by their *characteristic functions*

$$f_A(u) = \begin{cases} 1 & u \in A \\ 0 & u \notin A \end{cases} \tag{1}$$

A Boolean encoding of the states in a Kripke structure makes $f_A(\cdot)$ a Boolean function and any set operations now become Boolean operations. That is, set intersection becomes conjunction and set union becomes disjunction. Binary decision diagrams (BDDs) are efficient data structures for Boolean functions. Representing transition relations such as $R(x, x')$ makes it possible for MC to verify properties in systems with over 10^{100} states—far, far more states then what is found in realizable control strategies.

Several important issues concerning our use of model checking to check safeness of control strategies are worth highlighting:

- Model checking has been widely used to verify hardware and software systems. However, the large number of states often forces one to use a reduced FSM model, created via model reduction techniques, in which some details are abstracted out. This has important consequences: model checking may verify the reduced model is safe with respect to the operational environment, but this does not necessarily guarantee the original system is safe.

  However, in our approach the evolutionary algorithm renders FSMs which are complete in the sense that every aspect of the control strategy is explicitly described in the FSM structure. In other words, no details are abstracted out or reduced. *Consequently, in our application model checking will guarantee whether or not the candidate control strategy fully satisfies the safety properties.*

- Model checkers typically provide trace information to help pinpoint where the safety property failed.

  We will not use this feature. Indeed, we treat the entire safety issue as a decision problem—i.e., either the strategy is safe or it is not. Unsafe control strategies are immediately discarded, so there is no need to know why it is unsafe.

- Model checkers are used to verify functional specifications and other properties, e.g., liveness.

  In our approach model checking only verifies safety, which is simpler than trying to verify liveness. Any other performance criteria will be assessed by trying out the control strategy in its operational environment. This has an important consequence: $\mathbf{AG}(\cdot)$ is the only form of CTL operator we will ever need.

- In practice, control strategies tend to have orders of magnitude less states than what has been described above. Since model checking algorithm complexity is linear in the size of the FSM and in the length of the CTL expression [12], the safety of a control strategy can be quickly verified.

There are several very good model checkers available free from universities. Among these are SMV from Carnegie-Mellon University [15] and VIS [16, 17] from the University of California at Berkeley.

4 Implementation Details

This section provides detailed guidance on how to implement our method for evolving safe control strategies. An example problem is included to illustrate our approach.

4.1 A Test Problem

The easiest way to understand how to implement our approach is to explain it within the context of a problem. We will use a modified form of the *Santa Fe Trail Problem.* The unmodified Santa Fe Trail Problem, which is fully described in [18], involves placing an artificial Ant on a 32×32 grid. The Ant starts out facing east. At each time step the Ant can turn left, turn right, walk one step forward, or do nothing. Food pellets have been randomly scattered on the grid points and the objective is to have the Ant consume as much food as possible within a given number of time steps. The grid is the operational environment for the Ant and a control strategy tells the Ant how to circumnavigate the grid. Figure 4 shows how a control strategy is encoded in a FSM. Sanchez et. al [19] have recently shown that EAs can evolve FSMs that encode high performance control strategies for this problem.

As previously stated, the objective of our research effort is to see if our approach can quickly evolve safe control strategies. Consequently, we will need a modified version of the Santa Fe Trail Problem that adds a safety component. We call this new version the *Hazardous Santa Fe Trail Problem* because it introduces hazards—i.e., unsafe conditions—that the control strategy must avoid. These hazards are "black holes", which are randomly placed at vacant grid locations. If the Ant steps into a black hole, it dies. The FSMs we

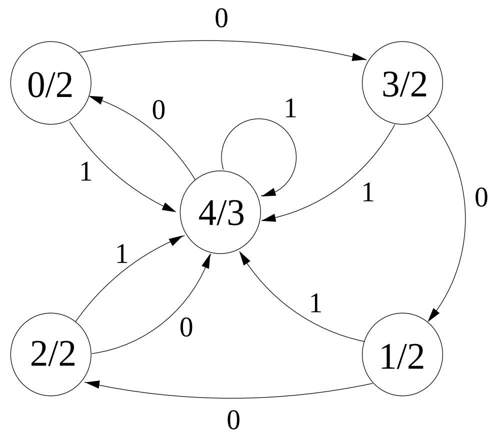

Figure 4: An example of an evolved FSM that encodes a control strategy for the Santa Fe Trail problem. The input to the FSM is an observation of food or no food and the output is a command for the Ant to make some movement. Any state may be chosen to be the initial state. The numbers on the arcs represent the observed input (0: no food, 1: food). Within the bubbles are indicated state/output action (0: NOP, 1: turn left, 2: turn right, 3: step forward). Every aspect of the control strategy is completely encoded within the FSM structure. For instance, if the current state is state 3, and no food is observed directly ahead, then the strategy transitions to state 1 and the Ant is directed to turn to the right. However, if food is observed directly ahead, the strategy transitions to state 4 and the Ant is directed to step forward to consume the food pellet. This control strategy is taken from [19]. The FSM for the Hazardous Santa Fe Trail problem is similar in structure, but the labeling of the arcs is different (see text).

evolve will be similar in structure to that shown in Figure 4, except now each arc is labeled with two observed inputs: one indicates the presence of food and the other indicates the presence of a black hole[1].

The Hazardous Santa Fe Trail problem is an ideal forum for evaluating not just our approach, but any method for designing safe adaptive control strategies. Observe that the currently implemented control strategy may be optimal for the existing placement of food pellets or black holes, but any changes in those locations can make the strategy ineffective (or even unsafe), thereby forcing an adaption. It is therefore easy to construct scenarios requiring adaption of an existing control strategy. Moreover, the problem is easy to simulate.

[1]We also dropped the NOP action.

4.2 Implementing the Evolutionary Algorithm

We intend to use random mutations as the reproduction operator in our EA. A number of mutation varieties are possible: adding or deleting states; changing the location of an arc; changing the action taken in a state; and changing the machine's initial state. The general rule for mutations is that they preserve some degree of similarity with the original structure, which leads to producing new structures with a similar fitness. This prevents the mutation process from degenerating into a simple random search (e.g., as done by simulated annealing).

The EA steps are shown in Figure 2. (There is one additional step, which is given in Section 4.3.) The algorithm begins with an initial population of μ randomly generated FSM structures, each encoding a control strategy. All μ FSMs are then evaluated for fitness by placing an Ant on the grid at location (0,0) and then executing the encoded control strategy for 200 time steps to see how many food pellets are consumed. The fitness value equals the number of food pellets consumed.

Subsequent iterations of the EA select μ parents, copy them, and then mutate the copies to produce new offspring FSMs. Only one of the above mutation operators is used to produce an individual offspring. (Mutation operators are chosen with equal probability.) The μ parents and μ offspring are collected into a temporary population. All of the individuals are ranked by fitness and the top μ survive while the rest are discarded. Notice that parent FSMs and children FSMs compete equally for survival. This iterative procedure, called a processing a *generation*, is repeated until a defined termination criteria is met. Usually this criteria is either an acceptable FSM has been found or a fixed number of generations have been processed. In the latter case the best fit FSM from the final generation is used.

The μ parents selected for reproduction are picked from the survivors of the previous generation. The highest fit individual from the previous population is copied unchanged to the next generation. This elitist policy ensures the fitness monotonically increases through-out the evolutionary process. The remaining $\mu - 1$ parents used during a generation are selected by conducting a binary tournament, which helps to choose the better fit parents for reproduction. Two randomly chosen parents compete in this tournament and there are two criteria used to determine the winner: first is best fitness and second is the best exploration capability[2]. If neither parent is a clear winner, then one of them is chosen at random. The binary tournament algorithm for the Santa Fe Trail Problem is described in Figure 5. The selection criteria, along with their priority, can be tailored for the problem at hand.

```
Randomly chose two parents $\alpha$ and $\beta$ from the
    current population

If fitness($\alpha$) > fitness($\beta$),
        then return $\alpha$
If fitness($\beta$) > fitness($\alpha$),
        then return $\beta$
If $\alpha$ has more states with action == 3,
        then return $\alpha$
If $\beta$ has more states with action == 3,
        then return $\beta$
If none of the above are satisfied, then return
    return $\alpha$ or $\beta$ with equal probability
```

Figure 5: A binary tournament algorithm. Each if-then statement is evaluated in the order shown.

It is important to start out with an initial population that is safe. This is easily accomplished even though that population is randomly generated. For example, in the Hazardous Santa Fe Trail problem, randomly assign the arcs between states, but do this in a way that ensures the arcs traversed whenever a hole is present do not point to a state with action = 3 (i.e., take a step forward).

These control strategies are intrinsically evolved so the only evaluation method is to download each of them into the system—in this case the artificial Ant—and try it out in

[2]For the Hazardous Santa Fe Trail Problem this latter characteristic is measured by the number of states who's action is to take a step forward.

the real physical environment; the better the strategy performs, the higher its fitness value. However, each strategy is first checked for safety and any unsafe strategy is immediately discarded and a new candidate strategy is evolved by mutating the same parent FSM again.

4.3 Adding MC to an EA

The model checker makes successive sweeps through the FSM states, labeling states in which the safety property holds. For example, consider the safety property $\mathcal{S} \Rightarrow$ *did not step into a black hole*. The CTL formula $\mathbf{AG}(\mathcal{S})$ says it is not possible to get to a state where $\mathcal{S}$ no longer holds, because this is unsafe condition for the Ant. All states are checked and labeled to indicate whether $\mathcal{S}$ holds. It is then possible to verify if the CTL formula is true or false in linear time. For the Hazardous Santa Fe Trail problem the formula for $\mathcal{S}$ is quite simple: never take a step forward if a hole in front of you. That is,

$$\mathcal{S} \Rightarrow \neg(\text{step forward} \wedge \text{hole present}) \tag{2}$$

The safety of a control strategy for the Hazardous Santa Fe Trail Problem can be checked in a straightforward manner. First, we note that each state has three incident arcs: one traversed if there is no food or no hole present; one if food is present but no hole is present; and one if no food is present but a hole is present. The MC process begins by initially labeling all states as safe. Next, check all states with action = 3 to see if they have an arc pointing to it that is traversed because a hole is present. All states incident to the tail of those arcs have their labels changed to unsafe. Following the procedure outlined in Figure 3, all states that transition to these unsafe states also have their labels changed to unsafe. This process continues until all states are visited and labeled as safe or unsafe. Finally, let $f_I(\cdot)$ be the characteristic function for the set of all initial states, and let $f_U(\cdot)$ be the characteristic function for the set of all states labeled as usafe. $\mathbf{AG}(\mathcal{S})$ holds if $f_I(u) \wedge f_U(u) = 0 \ \forall \ u$.

In the general case several safety properties will have to be satisfied. Similar sweeps must be conducted for every other CTL formulas describing safety properties, because all safety properties must hold before the control strategy can be deemed safe. This means the safety check for any control strategy is expressed in compact form as

$$\mathbf{AG}\left(\bigwedge_{j=1}^{K} \mathcal{S}_j\right) = 1 \qquad (3)$$

where $\mathcal{S}_j$ is the j-th safety property and $\mathcal{S}_j = 1$ means that property holds. Eq. (3) must hold from any initial state in the Kripke structure.

Any control strategy that fails the safety check is immediately discarded and the parent FSM is mutated again. This process usually will not have to be repeated too many times before a safe offspring is produced. Once the control strategy has been safety certified, it can then be evaluated in the operational environment. Hence, step 3 in Figure 2 should be changed to the following form:

3. create μ new FSMs by repeating the following two steps:

   (i) conduct a binary tournament to select a parent FSM from the current population

   (ii) randomly mutate the parent FSM to create a new offspring FSM

   (iii) use MC to verify offspring safety. if unsafe, go to step 3(ii).

5 A Numerical Example

The numerical example of the Hazardous Santa Fe Trail Problem was conducted on a 32×32 grid. Seventy food pellets and fifteen black holes were randomly assigned to grid locations (see Appendix for the exact locations).

Our EA used a population size of $\mu = 25$ FSMs that evolved over 150 generations. The initial population was randomly created with each FSM having between 10 and 15 states. Parents in subsequent generations were selected using a binary tournament. The mutation operators, defined in Section 4.2, were applied with equal probability but only one operator

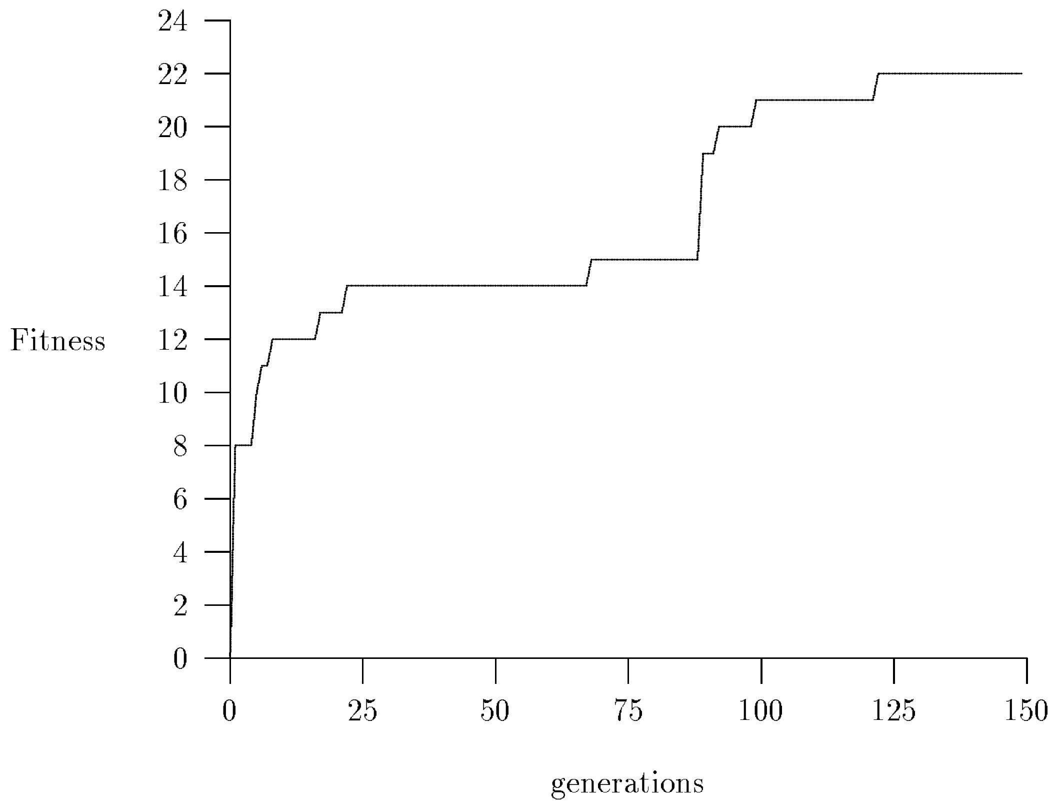

Figure 6: Fitness vs generations for a typical run.

could be used to produce a given offspring. Each offspring was safety checked using a MC algorithm before being evaluated for fitness. About 10%-15% of the created offspring were found to be unsafe. Recall FSMs that fail the safety check are immediately discarded and the same parent FSM is mutated again. This procedure must continue until a safe offspring is produced. Usually only one or two tries were sufficient to achieve this.

Each safe FSM was evaluated by placing the Ant at grid location (0,0) and executing the encoded control strategy. The number of food pellets the Ant consumed within 200 time steps was recorded and this became the fitness value of the FSM. The simulation was written in C++ and run on a SPARC ULTRA-10 workstation. Each run took less than 20 seconds to complete. Figure 6 shows the results of a typical run. Notice that the fitness monotonically increases. (See Appendix for the state table describing this FSM.)

It is important to realize that unsafe control strategies do not necessarily always produce unsafe results. We took one of the unsafe FSMs and executed it. (See Appendix for this FSM's state table). The Ant only consumed two food pellets before it died, which makes it a relatively poor control strategy. However, the Ant did not fall into a hole until time step 55—which means it was completely safe for 54 time steps. It is likely that this unsafe action would occur at some different time step with a different placement of food pellets and holes. But, the key point here is unsafe actions take place only if the FSM is in a specific subset of states and, even then, only if a specific set of inputs are present. A FSM could, in principle, undergo thousands of state transitions without ever producing unsafe actions. This means unsafe actions are eventuality events—something difficult to find using simulation or emulation methods. The MC algorithm checks for these unsafe eventuality events by identifying paths that terminate at unsafe states.

6 Final Remarks

The evolved control strategy is only good for a specific operational environment. Put another way, an evolved control strategy that is good for one operational environment may not be any good in a second environment because it's efficacy was tied to the first operational environment. The operational environment for an instance of the Hazardous Santa Fe Trail Problem is defined by the food pellet and hole placements. While it is true a safe control strategy for one placement of food pellets and holes will certainly be safe for any other placements, it's ability to consume food pellets in different environments may vary considerably. So, how useful is it to evolve such a restricted control strategy?

We believe restricted control strategies are the norm for real-world applications. Electronic systems installed in space vehicles are designed to optimally perform over specified environmental ranges. Any deviation from these range boundaries at a minimum degrades the system's performance or, in the worst case, leads to system failure. The control strategy should start to evolve so that a system can recover most (if not all) of its previous functionality. For instance, suppose a deep-space probe suddenly encounters a high radiation environment, which causes degraded performance in a gyro system. The gyro system's control strategy should begin to adapt, but this adaption is to a known environmental change. The control strategy does not have to make the gyro system work in say a high temperature or high pressure environment because nothing indicates such an environment exists. In other words, a changed operational environment invokes the evolutionary process, and the scope of that change is known. If this was not the case, then the environmental change goes undetected and there is no reason to suspect the existing control strategy is inadequate. In either case the control strategy is restricted to work under a defined operational environment.

No attempt was made to use implication tables or other reduction techniques to check for equivalent states in the evolved FSMs. These techniques may be appropriate for extrinsic evolution, but they are probably too computationally expensive for intrinsic evolution, which is the intended application area for our approach.

Finally, in this initial effort we only concentrated on the efficacy of the approach without worrying about computational effort. Clearly time to evolve cannot be ignored because an autonomous space vehicles cannot survive for an indefinite period of time without a viable control strategy. Hardware-only implementations of EAs have been developed [20], and we recommend this as the preferred method of implementation for deep-space probes because it is timely and fully supports *in-situ* intrinsic evolution. Our future efforts will focus on that very approach for evolving control strategies under real-time constraints.

Acknowledgement

The author thanks Professor Xiaoyu Song, an expert in formal verification methods, for his assistance with incorporating MC into the EA.

References

[1] Europe and NASA set new Cassini-Huygens plan. *JPL News Release*, June 29, 2001.

[2] A. Stoica, A. Fukunaga, K. Hayworth, and C. Salazar-Lazaro. Evolvable hardware for space applications. *Proc. ICES98*, LNCS 1478:166–173, 1998.

[3] D. Bernard, R. Doyle, E. Riedel, N. Rouquette, and J. Wyatt. Autonomy and software technology on NASA's Deep Space One. *IEEE Intelligent Sys.*, 14(3):10–15, 1999.

[4] J. Burch, E. Clarke, K. McMillian, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information & Computation*, 98(2):142–170, 1992.

[5] A. Frasier. Simulation of genetic systems by automatic digital computers, I. introduction. *Aust. J. Bio. Sci.*, 10:484–491, 1957.

[6] J. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Mich. Press, 1975.

[7] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution.* New York: John Wiley, 1966.

[8] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution.* Frommann-Holzboog (Stuttgart), 1973.

[9] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.*, 1(1):67–82, 1997.

[10] D. Keymeulen, A. Stoica, J. Lohn, and R. Zebulum (Eds.). *Proc. 3rd NASA/DOD Workshop on EHW.* IEEE Computer Soc., 2001.

[11] P. Angeline, G. Saunders, and J. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Trans. Neural Nets*, 5(1):54–66, 1994.

[12] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. Comput.-Aided Des.*, 13(4):401–424, 1994.

[13] A. Gupta. Formal hardware verification methods: a survey. *J. Formal Methods Sys. Des.*, 1(2/3):151–238, 1992.

[14] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.

[15] http://www-2.cs.cmu.edu/ modelcheck/code.html.

[16] R. Brayton et. al. VIS: a system for verification and synthesis. *Proc. Int'l Conf. Comput.-Aided Verification*, LNCS 1102:428–432, 1996.

[17] http://www-cad.eecs.berkeley.edu/ vis/.

[18] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[19] E. Sanchez, A. Perez-Uribe, and B. Mesot. Solving partially observable problems by evolution and learning of finite state machines. *Proc. ICES2001*, LNCS 2201:267–278, 2001.

[20] B. Shackleford, G. Snider, R. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura. A high-performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Prog. & Evol. Mach.*, 2(1):33–60, 2001.

Appendix

item	**location**
food pellets	(2 , 1) (2 , 23) (3 , 10) (3 , 15) (3 , 20) (3 , 23) (3 , 29) (4 , 4) (5 , 15) (5 , 31) (6 , 7) (6 , 14) (6 , 19) (7 , 7) (8 , 20) (8 , 22) (8 , 28) (8 , 30) (9 , 8) (9 , 10) (9 , 17) (10 , 12) (10 , 13) (10 , 20) (11 , 20) (11 , 28) (12 , 25) (13 , 2) (13 , 4) (13 , 10) (13 , 27) (14 , 30) (15 , 22) (16 , 6) (16 , 20) (17 , 26) (19 , 1) (19 , 2) (20 , 25) (20 , 30) (21 , 4) (21 , 7) (21 , 10) (21 , 13) (21 , 26) (21 , 30) (22 , 1) (22 , 4) (22 , 16) (22 , 25) (23 , 12) (24 , 7) (24 , 8) (24 , 9) (24 , 21) (24 , 24) (24 , 29) (25 , 3) (25 , 9) (25 , 11) (25 , 31) (26 , 22) (28 , 5) (28 , 20) (28 , 28) (29 , 25) (30 , 12) (30 , 13) (30 , 19) (31 , 5)
holes	(5 , 2) (6 , 2) (7 , 31) (8 , 13) (9 , 14) (17 , 17) (18 , 21) (18 , 27) (20 , 1) (23 , 21) (24 , 16) (25 , 24) (26 , 12) (27 , 23) (28 , 25)

Table 1: This table shows the 70 food pellet and 15 hole grid locations used in the Section 5 example.

present state	next state			action
	F H	F H	F H	
	0 0	**0 1**	**1 0**	
0	12	5	1	1
1	18	5	8	3
2	19	9	1	1
3	10	2	1	1
4	16	5	1	1
5	1	13	1	1
6	2	7	1	2
7	4	4	23	2
8	9	6	8	3
9	12	5	10	2
10	8	4	1	3
11	6	22	8	1
12†	3	7	14	3
13	11	5	8	2
14	17	2	12	3
15	3	5	21	1
16	20	11	8	3
17	10	6	12	2
18	16	3	8	3
19	17	17	1	3
20	7	0	18	3
21	12	6	14	3
22	7	6	12	2
23	0	13	1	3

Table 2: This is the state table for the best evolved control strategy found during a single EA run. The next state is shown for the three possible food/hole conditions visible to the Ant ("0" implies absence, "1" implies presence). Actions 1, 2, and 3 are turn left, turn right, and step forward, respectively. '†' indicates the initial state. The control strategy is safe because F=0 and H=1 never causes a transition to a state with action = 3. This control strategy consumed 22 food pellets in 200 time steps.

present state	next state			action
	F H	F H	F H	
	0 0	**0 1**	**1 0**	
0†	6	5	1	2
1	7	5	8	3
2	1	9	10	1
3	7	2	1	1
4	1	5	1	1
5	1	9	1	1
6	2	7	1	3
7	4	4	1	2
8	2	6	11	3
9	6	2	8	2
10	4	5	10	3
11	1	0	10	3

Table 3: This is the state table for one of the unsafe control strategies. The next state is shown for the three possible food/hole conditions visible to the Ant ("0" implies absence, "1" implies presence). Actions 1, 2, and 3 are turn left, turn right, and step forward, respectively. '†' indicates the initial state. Notice that state 8 is unsafe because F=0, H=1 causes a transition to state 6, which has action = 3. This control strategy caused the Ant to fall into a hole at time step 55.