Automated Verification of Model-based Programs Under Uncertainty

Tazeen Mahtab, Gregory T. Sullivan, Brian C. Williams
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
tmahtab@mit.edu, gregs@csail.mit.edu, williams@mit.edu

***Abstract*— Highly robust embedded systems have been enabled through software executives that have the ability to reason about their environment. Those that employ the *model-based autonomy* paradigm automatically diagnose and plan future actions, based on models of themselves and their environment. This includes autonomous systems that must operate in harsh and dynamic environments, like deep space. Such systems must be robust to a large space of possible failure scenarios. This large state space poses difficulties for traditional scenario-based testing, leading to a need for new approaches to verification and validation.**

We propose a novel verification approach that generates an analysis of the most likely failure scenarios for a model-based program. By finding only the most likely failures, we increase the relevance and reduce the quantity of information the developer must examine. First, we provide the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. We incorporate uncertainty into the verification process by acknowledging that all such programs may fail, but in different ways, with different likelihoods. The verification process is one of finding the most likely executions that fail the specification. Second, we provide a capability for verifying executable specifications that are fault-aware. We generalize offline plant model verification to the verification of model-based programs, which consist of both a plant model that captures the physical plant's nominal and off-nominal states and a control program that specifies its desired behavior. Third, we verify these specifications through execution of the RMPL executive itself. We therefore circumvent the difficulty of formalizing the behavior of complex software executives.

We present the RMPLVerifier, a tool for verification of model-based programs written in the Reactive Model-based Programming Language (RMPL) for the Titan execution kernel. Using greedy forward-directed search, this tool finds as counterexamples to the program's goal specification the most likely executions that do not achieve the goal within a given time bound.

I. INTRODUCTION

Highly robust embedded systems have been enabled through software executives that have the ability to reason about their environment. Such systems must operate in harsh and dynamic environments, like deep space, and therefore must be robust to a wide combination of potential failures. Those that employ the *model-based autonomy* paradigm automatically diagnose and plan future actions, based on models of themselves and their environment. *Model-based programming* is an approach to developing embedded systems that can reason about and control their hardware using corresponding software models [WICE03]. A *model-based program* enables one to specify the desired state evolutions of the embedded system. It consists of a specification of behavior, known as a *control program*, and a representation of the physical plant's nominal and off-nominal states, known as a *plant model*; these are run on a *model-based executive*. Model-based systems have the ability to detect and respond to unanticipated failures on the fly. Therefore, they provide an increased assurance of reliability and *fault awareness*. However, such programs present a new challenge to verification. First, the large space of failure situations they handle poses difficulties for traditional scenario-based testing. Second, they are run on a complex execution algorithm. This leads to a need for new kinds of verification and validation [PS00].

Traditional verification efforts, such as the symbolic model checking of reactive programs [BCM+92], have focused on determining the correctness of embedded programs. However, in the real world, where embedded programs control real hardware, those systems are never guaranteed to succeed; they are more or less likely to succeed. We extend model-based system verification to the verification of model-based programs under uncertainty.

We propose a novel verification approach that generates an analysis of the most likely failure scenarios for a model-based program. First, we provide the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. We incorporate uncertainty into the verification process by acknowledging that all such programs may fail, but in different ways, with different likelihoods. Second, by verifying model-based programs, we provide a capability for verifying executable specifications that are fault-aware. A model-based system is never guaranteed to function correctly, since it always has some probability of not behaving nominally. Therefore, verification of these systems is a process of finding the most likely of these failure executions, rather than simply determining if there are any. Third, we verify these specifications through execution. We therefore circumvent the difficulty of formalizing the behavior of complex software executives by invoking the actual executive components.

We present the RMPLVerifier, a tool for verification of model-based programs written in RMPL for the Titan execution kernel. The *Reactive Model-based Programming Language (RMPL)* allows developers to perform high-level reasoning

on the behavior of a model. The RMPLVerifier uses greedy forward-directed search to analyze an RMPL model-based program based on a goal specification and a given time bound. It then presents, as counterexamples, the most likely executions that lead to failure i.e. non-achievement of the stated goal. We analyze the results of applying our verification approach to the Mars Entry scenario, a significant model-based program specifying the entry sequence for a lander spacecraft.

II. MOTIVATION

Our verification approach provides three capabilities. We examine and motivate each of these in turn.

The first contribution of our approach is the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. In the past, one created robust embedded systems by attempting to enumerate ahead of time all possible failures the system could encounter. These systems were limited by the ability of the software development team. If a system encountered a failure that had not been predetermined by the developers, possible because of the many complex interactions between the hardware and software and the environment, it could fail to react properly. For example, the failure of the Mars Polar Lander is thought to have occurred because of unexpected leg sensor readings as it attempted to land. The software erroneously concluded from these readings that the Lander had touched down on the surface and prematurely shut off the engines, leading to the loss of the craft. New intelligent systems have been created to address this problem [WICE03]. Rather than being preprogrammed with all failures, these systems have the ability to deduce if they are in a nominal or failure state and to respond accordingly. These systems are stochastic, as they maintain knowledge of the probability of being in a particular state at a given time. We provide a verification capability for such systems. By returning information on the likelihood of the program's executions, we incorporate uncertainty into the verification process. By showing only the most likely failure executions, the RMPLVerifier helps focus the systems engineer on the most vulnerable components of a system.

Our second contribution is the ability to verify executable specifications that are fault-aware. Synchronous programming languages, such as Esterel [BG92], were designed for writing software to control reactive systems. A synchronous programming language is characterized by logical concurrency, orthogonal preemption, multiform time, and determinacy, which have been shown to be necessary characteristics for reactive programming [BG92]. Synchronous programming seeks to provide executable specifications. An *executable specification* is a program that doubles as a specification about which one can prove properties and an executable implementation of that specification. This eliminates the need to write a specification and implementation separately [WICE03]. Model-based programming generalizes this approach to executable specifications that are *fault-aware* - they have knowledge of the behavior of the plant under failures as well as nominal situations. A model-based program, consisting of a control program and plant model, is a fault-aware executable specification of the desired behavior of a robust embedded system. The plant model is a representation of the hardware, including the nominal and faulty states it may be in. The control program directs the actions of the executive to progress the system through a sequence of intended states. The executive uses the plant model to generate a control sequence that achieves these intended states. The verification task for a model-based program therefore has two pieces. One may verify properties of the plant model alone, or one may verify the control program, given a correct plant model. Our work focuses on the latter, while previous work has focused on the former [PS00].

There has also been research on the verification of the diagnostic executive [HLP+00] [LP03]. That research attempts to formalize the behavior of the executive, and then prove that the executive, thus formalized, has various properties. Our research takes a different approach: as part of the verification process, we invoke the actual executive components that will be used at runtime. This has the advantage of avoiding any mismatch between the formalization and the actual software used, while the disadvantage, of course, is that we cannot use logical reasoning to infer properties of the executive.

The control program, by its nature, has a high-level goal. For example, this could involve carrying out a navigation procedure or maintaining a sub-system in a steady-state. We enhance the control program to include this definition of success in the form of a goal specification. The results returned by verification are counterexamples to this overall specification.

Our final contribution is the ability to verify these specifications through execution. To handle all possible failure scenarios, the reactive systems that we have described must consider an exponentially large state space. To achieve tractability, model-based executives consider a subset of the possible situations and solutions, by employing anytime algorithms. Due to this approximate inference, it is difficult to formalize the behavior of such systems correctly. In addition, changes to approximations made by the reactive system over time would render any formalization used by a verifier obsolete. We therefore generate our results by running the specification on the actual software executive. Our tool verifies programs written in the Reactive Model-based Programming Language (RMPL) using the Titan executive. Titan includes both a *control sequencer* and *deductive controller*. The control sequencer generates the sequence of intended states, while the deductive controller attempts to achieve them. An RMPL model-based program can have many different executions, which depend on the observations it receives, the time for which it runs, and the mode estimation algorithm used for diagnosis. Some of these executions will achieve the program's goal, and others will not. For instance, along one execution path, a camera may fail to take a picture, resulting in an unsuccessful

navigation procedure. The verification tool focuses on these unsuccessful execution paths. It explores the set of most likely executions over the specified number of program steps. It interfaces directly with the Titan executive and can thus easily accommodate updates to Titan.

We further motivate our verification approach with an example.

III. Example of Verification on a Model-based Program

Consider the problem of controlling a spacecraft system. A spacecraft has hundreds of different components that must interact in complex ways. At the same time, a spacecraft operates autonomously in an unpredictable environment, making it likely that there will be unexpected failures. These properties make it a good candidate for model-based autonomy. Figure 1 shows a Mars lander spacecraft. Figure 2 shows the RMPL control program [Ing03] specifying the desired behavior of such a spacecraft during an entry scenario. The program performs a series of actions in preparation for entering the atmosphere of Mars, such as turning on the engine and letting it heat to standby, changing the kind of navigation used, and properly orienting the craft. The program operates on a model of the spacecraft, which represents both nominal and failure scenarios.

Fig. 1. The Mars Polar Lander. *Courtesy NASA/JPL-Caltech.*

```
1  EntrySequence() {
2    Engine = Standby;
3    Nav = Inertial;
4     do {
5        always (Att = Entry-Orient),
6        when (Att = Entry-Orient)
7           donext (Lander = Separated)
8    }  watching (Entry = Initiated)
9  }
```

Fig. 2. The RMPL Control Program for the Mars Entry Scenario.

Since many failure scenarios are possible, a developer creating a model-based program for such a system would find it beneficial to be able to enumerate possible failure executions. The RMPLVerifier returns the most likely executions of a model-based program that do not lead to success. The tool tracks a set of most likely plant state trajectories over time, as shown in Figure 3. The figure shows a simplified trajectory that could be returned as a counterexample by the verifier. In this trajectory, the engine transitions from off to heating and then to a failed state.

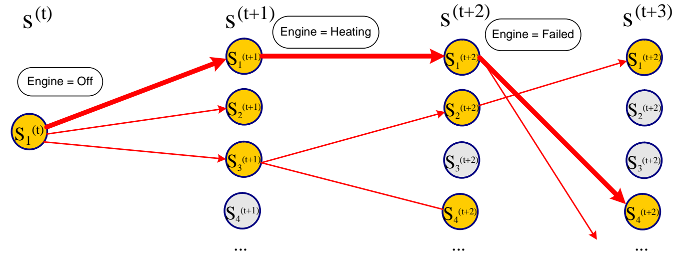

Fig. 3. The Set of Plant State Trajectories Tracked by the RMPLVerifier.

IV. The RMPLVerifier

The RMPLVerifier verifies a model-based program against a specification of success. It takes as input a model-based program with a goal specification and produces as counterexamples the k most likely failure executions of the program. The verifier generates these executions by searching the space of possible trajectories using greedy forward-directed search.

The RMPLVerifier generates trajectories using Titan and a simulator that provides observations consistent with the plant state. A *plant state trajectory* includes only states of the plant model and is tracked by the simulator. A *plant state* has a likelihood, computed from the likelihood of the previous state and the probability of transitioning to it from the previous state. A *program state* includes the states of the control program and plant estimate at a given point in the execution of a model-based program. A *program state trajectory*, which is generated by Titan, consists of a sequence of program states and represents the execution of a model-based program from the start state to a given time step for a given plant trajectory. The verifier returns a list of plant state trajectories. The likelihood of the plant state trajectories is used as the search heuristic. The search completes once the requested k number of solutions has been found for the given horizon. A list of trajectories, sorted in order of likelihood from highest to lowest, is returned as solutions to the verification query.

V. The Verification Algorithm

We seek to answer the verification question "What are the k most likely plant trajectories that do not achieve a given goal within N time steps, given the control program, plant

model, and starting configuration of an RMPL model-based program?"

Figure 4 gives an overview of the action of the RMPLVerifier. It has two main components, the Titan executive and the plant simulator. The Titan executive is the same software used to control the system at runtime; it consists of the control sequencer and deductive controller. It runs on the control program and plant model. The plant simulator tracks the set of k most likely trajectories at each time step, using the model and goal specification. The simulator and Titan interact in a loop. The simulator receives commands for the current time step from the executive and returns observations consistent with the next estimated state. At the end of a given time horizon, the simulator outputs the set of most likely plant trajectories at that time step that fail to achieve the program goal. This list, sorted in order of likelihood, is returned as counterexamples to the verification query.

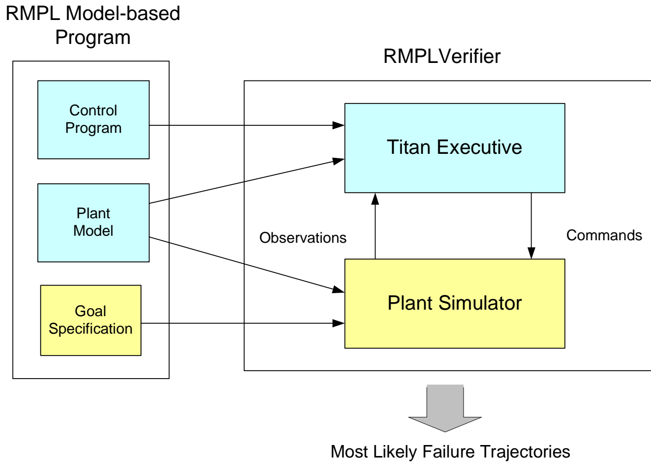

Fig. 4. A High-level View of the Algorithm.

A *plant state trajectory* includes only states of the plant model. A *plant state* has a likelihood, computed from the likelihood of the previous state and the probability of transitioning to it from the previous state. A *program state* includes the states of the control program and plant estimate at a given point in the execution of a model-based program. A *program state trajectory* consists of a sequence of program states and represents the execution of a model-based program from the start state to a given time step for a given plant trajectory. The verifier returns a list of plant state trajectories.

VI. Top-level Pseudocode

We first describe the top-level pseudocode for the verifier, shown in Fig 5.

We can think of the Simulator in abstract terms as tracking a set of plant state trajectories selected from the Trellis diagram of possible trajectories. This set of plant trajectories at first contains only the initial plant state. Titan, on the other hand,

```
1   Verify ( ControlProgram, Model, GoalSpec, InitialState,
2            Horizon, NumSolns) returns Trajectories
3
4   let Simulator = new Simulator( Model, GoalSpec,
              InitialState, NumSolns)
5   let InitObservation = compute assignment to observables
6        entailed by Simulator's initial state and Model
7   let Observations = { InitObservation }
8   let Commands = { }
9   for ( let TimeStep = 0; TimeStep < Horizon; TimeStep++ )
10    for ( let i = 0; i < Observations.size(); i++)
11      let Titan = get Titan that corresponds to plant trajectory i
12       let Command = Titan.Step( ControlProgram,
              Model, Observations[ i])
13      insert Command into Commands
14    endfor
15    Observations = Simulator.Step ( Commands )
16  endfor
17  return Trajectories from Simulator
```

Fig. 5. The Top-level Pseudocode of the Verification Algorithm.

can be thought of abstractly as generating a control program trajectory. The task of the verifier is to select amongst choice points in the trellis diagram, so that the simulator and Titan most likely fail to reach the goal.

In the beginning we initialize an instance of the simulator (Line 4). We next ask the simulator for the observation entailed by the initial plant state (Line 5-6). We add this initial observation to the list of observations to be passed to Titan (Line 7).

We now begin the first iteration of the verifier (Lines 9-16). We get an instance of Titan that corresponds to the simulator plant trajectory (Line 11). Since we are just starting, this will be a new Titan instance. We give our initial observation from the simulator to Titan (Line 12), as shown in Figure 6(a). Mode Estimation takes the observation and calculates an estimate of its initial state. The control sequencer generates the next configuration goal, based on this current state and the control program. Mode Reconfiguration issues a command for the plant based on this configuration goal, as shown in Figure 6(b). This is added to the list of commands (Line 13). At this point, Titan's program trajectory is composed of just one state, consisting of the states of the control program and plant estimate, originating from the initial observation. The simulator receives the new command from Titan (Line 15) and generates a new set of k observations from its plant trajectories, as shown in Figure 6(c).

We begin the second iteration. For each new observation, Titan is called for another step (Lines 10-14). The instance of Titan that we use must correspond to the plant trajectory for the observation (Line 11). In other words, the observation is output from a simulator plant trajectory, which generated a program trajectory using Titan in the previous iteration of the verifier. During the step, Titan's Mode Estimation learns of the new command from MR, as well as the new observation received from the simulator, and calculates the next estimated state

from the current state (Figure 6(d)). At this point, we have used Titan to generate k program state trajectories, obtained by extending the initial program trajectory with the different observations. We get a list of k new commands and pass them to the simulator (Line 15). The simulator ensures that each command is given to its originating plant state trajectory. This is the plant trajectory that generated the observation that in turn generated the command. The plant trajectories are extended once more, and the new observations are generated, as shown in Figure 6(e).

In this manner, the cycle repeats until the time horizon has been exceeded. The RMPLVerifier returns the set of most likely plant trajectories from the Simulator at the end of the last time step. Figure 7 illustrates the steady-state relationship between Simulator and Titan trajectories. Titan's MR issues a command. The Simulator receives that command and issues the new observation based on the next plant state. Titan's ME receives the observation as well as the command and computes its next state estimate.

We look into the operation of the Simulator in detail in the next section. For now, it is sufficient to know that the Simulator extends its initial trajectory based on the command by the next most likely states to a set of most likely trajectories. For each next state, the Simulator outputs an observation entailed by it. Therefore, if there are k next states, there are k corresponding observations (Figure 6(c)).

VII. The Simulator

We now examine the Simulator in greater detail. The pseudocode given in Figure 8 shows a **Simulator()** constructor by which we initialize the Simulator and a **Step()** function. **Step()** is called on each iteration of the verifier with a list of commands and returns a list of observations. The Simulator maintains a set of the current most likely plant trajectories, which it updates at each time step. Its first action is to invoke our modified k Most Likely Trajectories algorithm (Line 6), described later on. It provides the algorithm with the current set of plant trajectories and commands, as well as the model and goal specification of the model-based program, and obtains the set of next most likely program failure trajectories. For each next trajectory, it computes an assignment to the observable variables of the model that is entailed by the last state of the trajectory and the model (Line 17). It inserts this observation into a list of observations (Line 19). Finally, it updates the current trajectories (Line 21) and returns the list of observations to the verifier (Line 22). Figure 9 graphically illustrates one step of the Simulator. The Simulator receives a list of commands 1 through k, which are passed to the k current trajectories. Based on the new information, it generates the k next trajectories and their corresponding observations.

In our algorithm, we assume that the plant model is determinate, and therefore, a state uniquely determines an observation. However, if the plant model is an indeterminate, partial specification, then multiple observations may be consistent with a state, and the observations may have different likelihoods. Therefore, branching on unassigned observable variables with different probabilities could be a future extension to this algorithm.

The Simulator uses a modified version of a k Best Trajectories algorithm originally developed for the Mode Estimation component of Titan. Figure 10 gives the pseudocode incorporating our changes to the algorithm. Since we are using Titan to individually propagate a set of trajectories rather than just one, we receive a corresponding number of commands. Observations are not part of the input, since our objective, as a simulator, is to generate the observations from the commands. Finally, we consider the goal specification of the model-based program when determining which transitions are tracked. Since we are interested only in counterexamples, that is, trajectories that do not satisfy the goal specification, we disallow transitions to states that fulfill the goal of the program. The goal specification is a propositional state logic sentence; for example, a specification could be (**RoverTargetPosition** = **Reached**). We take the negation of this sentence and add it to the logic constraints for the model. We determine enabled transitions based on whether they satisfy the negation of this goal constraint, in addition to the constraints asserted by the modes and transitions of the model. Therefore the algorithm only generates the most likely trajectories that fail to achieve the program goals. The cost of a trajectory is its computed probability and therefore, the probability of the execution it represents. The algorithm performs a greedy forward-directed search over the space of possible trajectories. At each iteration, best-first search is used to select the next set of trajectories.

VIII. Results and Future Work

The RMPLVerifier has been implemented in C++ and integrated with the current version of Titan and RMPL.

We measured the performance of the RMPLVerifier with respect to time. The program was tested on an Intel(R) Xeon(TM) 1.7 GHz processor with 500 MB of RAM running the Debian Linux 2.4.23 operating system. It was run on the Mars Entry model-based program with different combinations of search parameters. The Mars Entry model is composed of 8 component models. It has 7 control variables, 15 observable variables, 10 state variables, and 10 dependent variables, for a total of 42 variables. It has 179 transitions.

For our first benchmark, we varied the number of time steps the verifier examined while keeping the number of tracked trajectories constant. Table I shows the time performance of the program in this case. For each test, we measured the total number of seconds that the process used directly in user mode and the total number of seconds used by the system on its behalf in kernel mode. We added these two numbers to obtain the total time the process ran. We averaged the total time over ten runs to compute the average time for the test case.

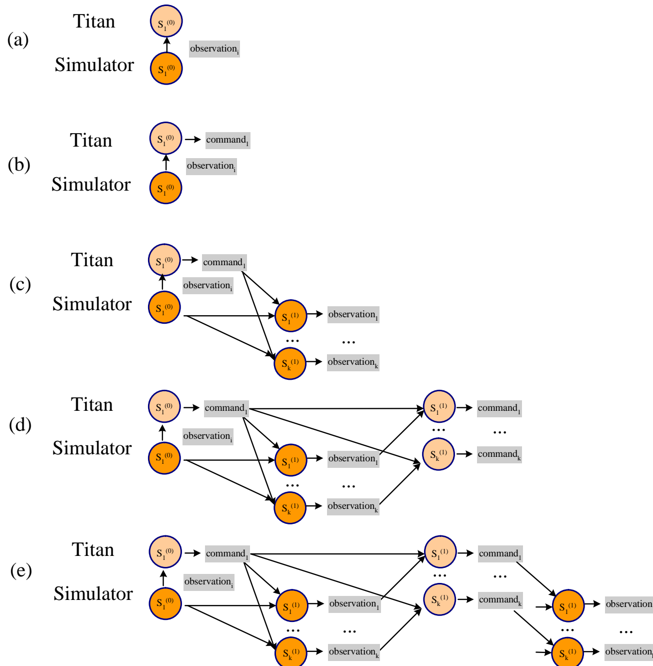

Fig. 6. The Beginning of the Verification Algorithm. (a) The Simulator issues an observation based on the initial state. (b) Titan estimates the state based on the observation and issues a command. (c) The Simulator issues observations consistent with the next states. (d) Titan issues a new set of commands based on the new observations. (e) The Simulator again issues observations consistent with the next states.

Number of Time Steps	Number of Trajectories	Average Time (seconds)
1	5	0.396
2	5	2.898
3	5	5.96
4	5	10.066
5	5	15.758
6	5	22.462
7	5	29.511

TABLE I
PERFORMANCE OF RMPLVERIFIER ON THE MARS ENTRY MODEL-BASED PROGRAM WITH RESPECT TO THE NUMBER OF TIME STEPS.

Number of Time Steps	Number of Trajectories	Average Time (seconds)
1	1	0.349
1	2	0.36
1	3	0.363
1	4	0.388
1	5	0.412
1	6	0.426
1	7	0.441

TABLE II
PERFORMANCE OF RMPLVERIFIER ON THE MARS ENTRY MODEL-BASED PROGRAM WITH RESPECT TO THE NUMBER OF TRAJECTORIES.

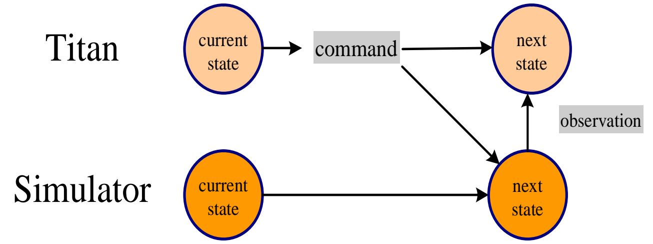

Fig. 7. Relationship between a Titan trajectory and a Simulator trajectory.

```
1  Simulator ( Model, GoalSpec, InitialState, NumSolns ) {
2
3    // Constructor
4    Simulator. Model = Model
5    Simulator. GoalSpec = GoalSpec
6    Simulator. NumSolns = NumSolns
7    Simulator. CurrentTrajectories = initial trajectory from InitialState
8  }
9
10  Step ( Commands ) {
11  returns Observations
12
13    let NextTrajectories = FindModifiedKMostLikelyTrajectories
14      ( Simulator. Model, Simulator. GoalSpec, Simulator. CurrentTrajectories,
15        Commands, Simulator. NumSolns )
16    foreach NextT in NextTrajectories
17        let Observation = compute assignment to observables entailed by NextT's
18        current state and Model
19      insert Observation into Observations
20    endfor
21    Simulator. CurrentTrajectories = NextTrajectories
22    return Observations
23  }
```

Fig. 8. The Simulator Pseudocode.

For our second benchmark, we varied the number of trajectories to find while keeping the number of time steps constant. Table II shows the time performance of the program. We computed the average time in the same manner.

The data we collected agreed with our intuition on the performance of the program. As we increase the number of trajectories, the time for the program to complete increases in a roughly linear fashion. This corresponds to the behavior we would expect for Mode Estimation, which is the computational core of the simulator. However, as we increase the number of time steps, the time for the program to complete increases exponentially. This is an artifact of the implementation. We can understand why this happens by looking at the process by which the verifier computes the next set of Titan trajectories from the current one. Let us define the time needed to extend each of the current set of Titan trajectories by one Titan step as a constant T. Therefore, the time taken to complete one time step is T added to the time to recreate the current Titan trajectories. This can be described by the following recurrence:

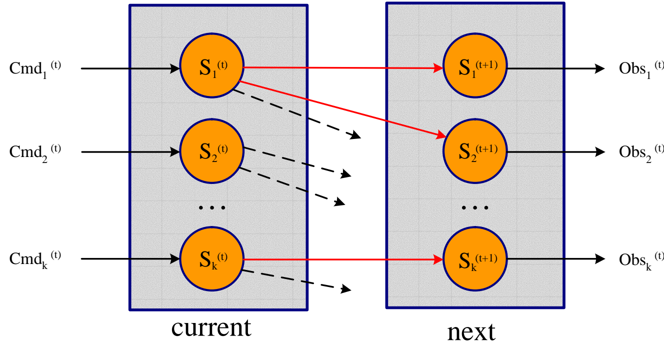

Fig. 9. One step of the Simulator.

```
1  ModifiedFindKMostLikelyTrajectories ( Model, GoalSpec,
2        CurrentTrajectories, Commands, NumSolns )
3   returns NextTrajectories
4
5  let NextTrajectories = {}
6  let PriorityQueue = {}
7  foreach CurrentT in CurrentTrajectories
8     let Command = Commands[ CurrentT ]
9     compute the most likely transition from CurrentT's
         current state, enabled by Command and Model,
10        that doesn't satisfy GoalSpec
11     let NextT = CurrentT + target state of enabled transition
12     insert NextT into PriorityQueue
13  endfor
14  while PriorityQueue is non-empty
15     let T = pop most likely trajectory from PriorityQueue
16     insert T into NextTrajectories
17     if ( size of NextTrajectories == NumSolns )
18       return NextTrajectories
19     endif
20     let OrigCurrentT = T - last state of trajectory T
21     let Command = Commands[ OrigCurrentT ]
22    compute the next most likely transition from OrigCurrentT's cur-
rent state,
23     enabled by Command and Model, that doesn't satisfy GoalSpec
24     let NextT = OrigCurrentT + target state of enabled transition
25     insert NextT into PriorityQueue
26  endwhile
27  return NextTrajectories
```

Fig. 10. The Modified k Most Likely Trajectories Algorithm.

$$Step_n = T + Step_{n-1} + Step_{n-2} + ... + Step_1 = T * 2^{n-1}$$

Therefore, the time to execute a program that examines n steps is:

$$Time_n = \sum_{i=1}^{n} Step_i = T * (2^n - 1)$$

This performance bottleneck is strictly an implementation issue, which can be addressed as a software engineering task.

In addition to addressing performance issues, there are two main ways that the presentation of results to the user could be improved. The first is the addition of a graphical user interface in addition to the command-line interface that is currently available. Desired features of this interface could include the ability to visualize, step through, and play back trajectories. Another is the abstraction of the trajectories returned. It is possible that some trajectories returned will have many states in common. Trajectories could be grouped together by similarity and common points of failure to increase the relevance of the results to the user.

References

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BG92] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, Jan 1992.

[HLP+00] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

[Ing03] Michel D. Ingham. *Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences*. PhD thesis, Massachusetts Institute of Technology, May 2003.

[LP03] A. E. Lindsey and Charles Pecheur. Simulation-Based Verification of Livingstone Applications. In *Proceedings of Workshop on Model-Checking for Dependable Software-Intensive Systems (DSN 2003)*, San Francisco, CA, June 2003.

[PS00] C. Pecheur and R. Simmons. From Livingstone to SMV: Formal Verification of Autonomous Spacecrafts. In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS)*, Greenbelt, MD, April 2000.

[WICE03] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *IEEE*, 9(1):212–237, Jan 2003.