Analysis and Testing of PLEXIL Plans

Jason Biatek, Michael W. Whalen, Mats P. E. Heimdahl, Sanjai Rayadurgam
Department of Computer Science & Engineering
University of Minnesota
[biatek,whalen,heimdahl,rsanjai]@cs.umn.edu

Michael R. Lowry
Intelligent Systems Division
NASA Ames Research Center
Michael.R.Lowry@nasa.gov

ABSTRACT

Autonomy is increasingly important in missions to remote locations (e.g., space applications and deep sea exploration) since limited bandwidth and communication delays make detailed instructions from a remote base impractical. The planning systems used for autonomous operation are difficult to verify and validate because they must create plans for use in a specific environment and the correct behavior might not be easy to define.

To explore verification and validation of planning systems, we have developed a verification framework for the PLEXIL plan execution language. The framework allows us to perform verification and test case generation using Java Symbolic PathFinder. Using this framework, we have performed verification, bounded verification and test-case generation for NASA-relevant PLEXIL plans and discovered two bugs in the PLEXIL language semantics: one in the definition of the If/Then/Else construct in the Extended PLEXIL language that could lead to plan deadlocks, and one in the semantics of the core language that could cause the PLEXIL executive to crash.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Testing tools

General Terms
Verification

Keywords
Java, Java PathFinder, automatic test generation

1. INTRODUCTION

NASA has a strong interest in using autonomous systems in future space missions, due to inherent bandwidth and latency issues for spacecraft, and to support longer-term missions in which some component failures are to be expected. Systems that operate autonomously do so using a planning system, which must take into account the environment and the goals of the system and then produce a viable plan to achieve the goals. In a mission-critical system, it is vital that the plans that are produced achieve the intended goals while accommodating the constraints of the environment.

Verification and validation of planning systems is not well explored, and there are specific challenges not found in most other critical software. For example, the correct behavior for an autonomously generated plan may not be easy to define. Also, plans are generated for a specific environment, which must be included in the analysis. All of these factors add complexity to the task of verification and validation. This complexity currently limits the scale at which formal verification can be performed to small plans. Therefore, we are also pursuing testing as a scalable complement to formal verification and are building a tool suite that uses symbolic analysis to generate test cases for plans. By automatically generating test suites that meet coverage criteria and ensuring that a plan meets its requirements in the tests, we can credibly argue that the plan is operating correctly in its intended environment.

The Plan Execution Interchange Language (PLEXIL) [5] is a high-level language developed at NASA to specify plan execution. PLEXIL models are constructed from nodes; nodes in PLEXIL are state machines that describe different aspects of system behavior. These nodes can be combined in parallel and hierarchical organizations to perform complex plan activities. In fact, the state machines can be used to implement standard programming language idioms such as alternation, iteration, and assignment, while also supporting a variety of specialized features to support plan execution including a type system extended with unknown values and a variety of abort mechanisms to interrupt plan execution when invariants are violated. PLEXIL uses a synchronous notion of time with execution progressing in discrete steps—all of the state machines logically execute in parallel at each step—and has a well-defined semantics [1].

To support rigorous verification of PLEXIL plans, we have built a translator from PLEXIL to Java and a system for automated test case generation using the Java Symbolic PathFinder (SPF) [4]. In this paper, we describe our initial translation and analysis results using our PLEXIL analysis tool. With our translation framework, we are able to generate rigorous test suites for a variety of models. In the process of generating tests and constructing our analysis tools, we have found two bugs in the PLEXIL language definition.

(c) 2014 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

http://dx.doi.org/10.1145/2593489.2593496

FormalISE’14 May 31 - June 07 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2853-1/14/05 ...$15.00.
one in the Extended PLEXIL macros and one in the formal semantics of the PLEXIL core language. These issues have been confirmed by the PLEXIL development team and corrected based on our feedback.

The remainder of the paper is structured as follows: Section 2 describes background information about Java Pathfinder and the PLEXIL language. Section 3 describes the translation from PLEXIL into Java for analysis using SPF. Section 4 describes an experiment showing the tool performance on a range of NASA examples. Section 5 describes language issues discovered as a result of analysis, Section 6 describes future work, and Section 7 concludes the paper.

2. BACKGROUND

To provide the necessary background for a self contained report, here is an overview of the analysis tool used – SPF – as well as the language of interest — PLEXIL.

Java Pathfinder.

Java Pathfinder [2] is a model checker for Java programs that supports both explicit-state and symbolic model checking. As a concrete model checker, a program can obtain input values for analysis through method calls such as getBoolean(). Under a normal JVM, this will return a random boolean value. But when run under JPF, the value is nondeterministic, so both the true and false branches will be explored. JPF implements its own JVM and controls the execution of the target program at the byte code level, and implements state matching so that it can avoid executing the same path multiple times. By using this mode exclusively, one can be sure that all possible combinations of inputs have been explored, but the state space—of course—expands extremely quickly. The Symbolic Pathfinder package allows inputs to methods to be marked as symbolic, allowing symbolic evaluation of Java programs. This allows analysis of programs with large-domain numeric variables, but at the cost of completeness—it is not possible in general to know whether all system states have been explored. However, it is a promising technique for test-case generation, as each symbolic path explored can be “concretized” into a test case.

PLEXIL.

A PLEXIL plan consists of a hierarchy of nodes. Each node contains a state machine that controls its execution. A node will always start in the state Inactive and move through a progression of states like Waiting, Executing, Finishing, and end its execution in the Finished state. Nodes also have a set of conditions that are used as guards on many of the transitions in the state machine. For example, a node does not move from Inactive to Waiting until the node’s StartCondition has become true. There are also Preconditions, PostConditions, and InvariantConditions that are checked at various points during execution that will cause the node to fail if they are not satisfied. The state machine behavior is defined by a set of transition rules. Node state transition diagrams (such as Figure 2) describe the conditions under which a node changes from one state to another, as well as any actions performed.

Lookups, commands, the results of commands, and key-value pair updates from the PLEXIL plan constitute the input and output of the plan. A condition can include a Lookup expression to access information about the environment such as a sensor value. For example, in Figure 1, the Speak node’s StartCondition requires that the value LeverPulled is true. A Command node tells the executing environment to perform some action (possibly with arguments), and receives feedback from the environment. The Spin node executes the command SpinSelector when it executes. When passed to the environment, commands can be accepted, successfully run, rejected due to lack of resources, etc. Update nodes allow the PLEXIL plan to output key-value pairs to the executing environment, with no expectation of feedback.

The core PLEXIL language uses only a few different node types. Each node type has a slightly different state machine, but all behave in a similar fashion, with most performing a specific action upon entering the Executing state. Empty nodes do not perform a specific action when they execute, but can be referenced by other nodes to guide execution (for example, ensuring that an invariant holds while other nodes perform other tasks). Assignment nodes assign a value to a variable when they execute. Command nodes send commands to the environment. List nodes serve to hierarchically compose nodes and are the only nodes in core PLEXIL that contain other nodes. Library nodes are essentially List nodes, but the definitions of the child nodes are loaded from another PLEXIL file at the time the plan is loaded.

Nodes that end or fail cause their children to either finish or also end in a failure, but this is not done through any direct communication or message passing between nodes. Rather, the node transitions defined by PLEXIL are configured so that cascading effects occur through the child nodes monitoring the state of its ancestor nodes (see Figure 2).

Extended PLEXIL.

Although through careful specification of PLEXIL nodes one can implement loops, conditional branches, and other familiar programming constructs, it is cumbersome and tedious. Extended PLEXIL is a layer of syntactic sugar on top of PLEXIL that provides standard imperative programming language constructs: If/Then/Else, For loops, While loops, and Sequences. An example of an Extended PLEXIL plan can be found in Figure 3. The while loop repeatedly performs a sequence of: moving one meter, taking a picture, updating a counter, and printing status information. The final compiled PLEXIL plan replaces the while statement with nodes implementing the behavior of a while loop.

Formal Semantics of PLEXIL.

The formal semantics of PLEXIL [1] break the execution process down into different relations, each building on the
SafeDrive:

```java
{ Integer pictures = 0;
    EndCondition
    Lookup(WheelStuck) || pictures == 10;

    while (! Lookup(WheelStuck))
        Sequence
        { OneMeter: { Drive(1); }
            TakePic: {
                StartCondition pictures < 10;
                TakePicture();
            }
            Counter: {
                PreCondition pictures < 10;
                pictures = pictures + 1;
            }
            Print: { print ("Pictures taken: ", pictures); }
        }
}
```

Figure 2: The transition rules for nodes in the Waiting state [3]. PLEXIL uses a three valued logic, with an additional Unknown value for each data type.

Figure 3: An example PLEXIL plan illustrating some of the basic features of PLEXIL, including some of the Extended PLEXIL constructs.

next, as shown in Figure 4. The lowest level is expression evaluation, relating an expression to its value. The next layer is the atomic relation, which describes a node taking a single transition in its state machine (such as the ones in Figure 2). A micro step is defined as the result of all the nodes in a PLEXIL plan simultaneously taking one atomic step. From there, a macro step is defined as the repeated application of micro steps until a plan reaches quiescence (no state transitions occur in a micro-step) or an Assignment occurs or a Command is issued.

3. TRANSLATION TO JAVA

To generate tests from PLEXIL plans, a translator was written to convert a PLEXIL plan to Java code to support execution and take advantage of existing tool support for test case generation from Java programs. Our initial effort was a straightforward, naïve translator that created a class for every node, closely mirroring the structure of the original PLEXIL code and the specification of the PLEXIL semantics. A node extended a superclass for its type (Assignment, Command, List, etc) and contained much of the same information that was included in the original PLEXIL file. The translated class added a state machine step function and any variables that the node contained, allowing the superclass to implement the correct PLEXIL semantics while running the plan. Figure 5 shows an example of this code.

An object implementing the ExternalWorld interface is used by the translated PLEXIL plan to get information about the executing environment. It receives Commands and Updates, and returns Lookup values when the plan requests them. When the PLEXIL plan has reached a state where no more transitions are possible, the ExternalWorld is informed that the macro step has ended.

We have created a series of optimizations for the generated PLEXIL code in order to improve test-generation time in JPF, described in the subsequent sections. These optimizations substantially improve the analysis performance of the generated code in most cases, as described in Section 4.

3.1 Three Valued Logic Optimization

Making the representation of extended types efficient is crucial when optimizing the translated code. Recall that each type can take on the value Unknown. Initially, values were stored as simple classes that stored a boolean, number, or string, along with a boolean flag for determining whether the value was Unknown or not. Because each value is wrapped
in an object to accommodate Unknown, a substantial amount of object creation and cleanup is required. To reduce overhead, we created singleton objects for Unknown and boolean values, substantially reducing the number of objects created and reducing memory overhead for evaluation.

3.2 Constant Propagation

The conditions of a PLEXIL node have specified defaults that can be overridden for an individual node. For example, the StartCondition of a node is ‘true’ by default. Usually only a handful of the eight user-definable conditions are defined, and most of them have a default value. This means that many of the transitions defined in the PLEXIL semantics end up being guarded by constant values. We added a step where the translator propagates constants and removes transitions whose guard evaluates to the constant ‘false’. For example, when checking the expression foo || bar to see if it is true, an Unknown value is effectively biased towards either true or false. For example, when checking the expression foo || bar to see if it is true, an Unknown value is equivalent to false, since we are only interested in whether the statement is true. Since the three valued logic library for booleans includes methods such as isTrue(), isKnown(), isNotFalse(), etc., we can push this bias down to the leaves of the expression. Using the same example, when checking to see if the PLEXIL expression foo || bar is true, the emitted Java code will be foo.isTrue() || bar.isTrue() instead of (foo.or(bar)).isTrue(), reducing the evaluation time by using Java short-circuit evaluation.

3.3 Increased Use of Native Java Booleans

Boolean expressions in PLEXIL operate under a three-valued logic, requiring expressions to be evaluated using objects instead of the native Java analogues. However, most of the guards for transitions check to see whether a condition is true, not true, false, or not false. When this is the case, an Unknown value is effectively biased towards either true or false. For example, when checking the expression foo || bar to see if it is true, an Unknown value is equivalent to false, since we are only interested in whether the statement is true. Since the three valued logic library for booleans includes methods such as isTrue(), isKnown(), isNotFalse(), etc., we can push this bias down to the leaves of the expression. Using the same example, when checking to see if the PLEXIL expression foo || bar is true, the emitted Java code will be foo.isTrue() || bar.isTrue() instead of (foo.or(bar)).isTrue(), reducing the evaluation time by using Java short-circuit evaluation.

3.4 Lazy Evaluation

A significant amount of runtime was saved by being more selective about calling step functions for nodes. By design, children of Inactive nodes cannot do anything but remain Inactive. This property allows us to skip the step function for children of Inactive nodes, since they are guaranteed to be Inactive and to stay Inactive for at least one micro step. For a large plan, this could allow large sections of the node tree to be skipped during a step, and avoid executing the step function of every single node in the plan.

3.5 Dead Variable Removal

The PLEXIL semantics define a number of variables inherent to each node. The node’s state, outcome, and reason for failure are all included in each node and can be included in expressions. A feature of PLEXIL is time points, which allow a node to see at what time (defined by performing a Lookup of "time") a state was entered or exited. These are however rarely used and with each node containing 14 time point variables (two each for all 7 PLEXIL states), the number of variables quickly rose. By scanning all transition guards, assignments, commands, and updates, it is possible to see which time points were actually being read and remove any that were not. This dead variable removal resulted in a substantial reduction in the number of objects being created for each node. In fact, for one of our largest models, the “Fluid” model, this variable removal was necessary to even compile the generated Java code because the number of variables was exceeding Java bytecode limits.

3.6 Intermediate Language-Based Translation

Even with structural optimizations, a naïve approach that translates the PLEXIL semantics into Java inevitably leads to inefficiency. The Extended PLEXIL constructs are a convenient way to introduce imperative features, but they are still built out of core PLEXIL nodes that all run in parallel. This means that in a large plan, there could be hundreds or thousands of nodes, all running in parallel and checking for a possible transition each micro step, none of which will move because they are actually part of a structure that is intended to behave sequentially. To analyze PLEXIL models at scale, we must have a means for collapsing these parallel state machines into sequential state machines.

To perform significant optimizations that reorganize and combine nodes together, it is necessary to manipulate the PLEXIL plan in an abstract way. We have introduced an intermediate language to represent the PLEXIL plan before translating fully to Java to support substantial transformations and optimizations of PLEXIL plans.
Plan = {StateMachine }+ , { Variable } ;
StateMachine = { NodeId } , { State } ,
{ Transition } ;
NodeId = string ; (*unique ID for a PLEXIL node*)
State = { Tag } , (action) , "|" , (action) ;
(* Tags, entry actions, actions for each step *)
Transition = State , State , expr , (action) ;
(* Start state, end state, guard, actions *)
Tag = NodeId , PlexilState ;
Variable = NodeId , string , type , [expr] ;
(* ID, name, type, initial value *)
PlexilState = "Inactive" | "Waiting"
| "Executing" | "Finishing" | "Failing"
| "Iteration_ended" | "Finished" ;

Figure 6: PLEXIL IL Grammar

The grammar in Figure 6 shows the components in the intermediate language. A StateMachine in the IL represents a set of sequentialized PLEXIL nodes; we perform merging of PLEXIL nodes in order to reduce the analysis cost of the model. For each State in the StateMachine, there must also be a mapping back to a set of native PLEXIL states, since a PLEXIL node can query the state of other nodes. The behavior of the state machines is driven by Transitions and Actions. The Actions describe all of the possible behaviors of a PLEXIL node.

For example, in Figure 2, there are Actions associated with the transitions from Waiting to IterationEnded and Finished. Additionally, most node types perform their specific Action upon entry to the Executing state.

The intermediate language was designed to make manipulations of the StateMachines straightforward. The IL semantics are simply to perform a step in the root StateMachine, and execute any actions that occur. This format allows merging of parallel nodes in certain contexts, which will be described in Section 6. By merging parallel nodes, we significantly reduce the amount of code that must be executed in a microstep, which will reduce the analysis time substantially.

4. EXPERIMENTAL RESULTS

To assess the practical result of our optimizations, we present a comparison of the original naïve translator with the one using the intermediate language. The original does not have the three valued logic optimization, code specialization, lazy evaluation, or dead variable removal, and it was not compiled through the intermediate language. To measure the effectiveness of these optimizations on test case generation time, we translated PLEXIL plans using both techniques and ran them in SymbolicPathFinder with its included SymbolicSequenceListener enabled to generate test cases.

4.1 JPF/SPF Caveats

While analyzing the results of our test case generation runs, we noticed that many of the test cases generated by SymbolicSequenceListener were prefixes of longer test cases. Since a prefix of a test case is clearly made redundant by another test case that extends the same path, we wanted to remove the superfluous test cases. The SymbolicSequenceListener works by listening for either a backtrack or a property violation. In either case, it attempts to solve the current path condition and produce a test case, which is then saved to be printed at the end of execution. We added a check for prefixes before adding the test case so that these redundant test cases are not saved.

4.2 Analysis Timings

The PLEXIL distribution comes with a number of sample plans, such as CruiseControl, DriveToSchool, and SafeDrive. Our PLEXIL translator uses these examples in its regression test suite, and they are also useful as examples for test case generation.

The results of analysis can be found in Figure 7. By default, when generating tests using SPF, one sets a “depth” that describes the maximum number of symbolic choice points to be evaluated before considering the test case to be complete. For generation, we set the maximum time for generation for 20 minutes; test suites that required more than 20 minutes were marked as “timed out” (TO in the table). For these examples, the results are encouraging. The optimizations implemented in the new intermediate language result in significant reductions in execution time, while generating similar suites of test cases.

Figure 7: Results for SPF examples.

Another example that was analyzed was the Fluid model that is part of a larger model that describes portions of the International Space Station (ISS) environment. In the analysis of this model, we saw some anomalous performance data. The fully optimized intermediate language code takes longer to analyze than the original naïve version. We added

<table>
<thead>
<tr>
<th>CruiseControl</th>
<th>Depth</th>
<th>Naïve time</th>
<th>Naïve tests</th>
<th>IL time</th>
<th>IL tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>0:02</td>
<td>9</td>
<td>0:01</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0:07</td>
<td>70</td>
<td>0:04</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0:40</td>
<td>1240</td>
<td>0:15</td>
<td>1250</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>6:41</td>
<td>10788</td>
<td>2:32</td>
<td>10572</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DriveToSchool</th>
<th>Depth</th>
<th>Naïve time</th>
<th>Naïve tests</th>
<th>IL time</th>
<th>IL tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>0:21</td>
<td>9</td>
<td>0:05</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1:20</td>
<td>57</td>
<td>0:19</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>11:35</td>
<td>627</td>
<td>2:53</td>
<td>627</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>TO</td>
<td>TO</td>
<td>13:35</td>
<td>2715</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SafeDrive</th>
<th>Depth</th>
<th>Naïve time</th>
<th>Naïve tests</th>
<th>IL time</th>
<th>IL tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>0:03</td>
<td>7</td>
<td>0:02</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0:07</td>
<td>31</td>
<td>0:03</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0:44</td>
<td>255</td>
<td>0:15</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>2:18</td>
<td>1023</td>
<td>0:43</td>
<td>1023</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>17:13</td>
<td>8191</td>
<td>5:12</td>
<td>8191</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SimpleDrive</th>
<th>Depth</th>
<th>Naïve time</th>
<th>Naïve tests</th>
<th>IL time</th>
<th>IL tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>0:01</td>
<td>5</td>
<td>0:01</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0:02</td>
<td>23</td>
<td>0:01</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0:07</td>
<td>191</td>
<td>0:04</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0:21</td>
<td>767</td>
<td>0:12</td>
<td>767</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>2:42</td>
<td>6143</td>
<td>1:28</td>
<td>6143</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>19:32</td>
<td>24575</td>
<td>14:11</td>
<td>24575</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>
flags to the translator so that we could turn on and off individual optimizations in an effort to see what was causing the sudden performance drop. We discovered that for some reason, using the code specialization optimization (removing transitions that are never taken, either because their guard is false or a higher priority transition is unconditionally taken) results in much slower test case generation. Currently, we have no explanation for this behavior since the extra code present in the unoptimized code should never execute and, thus, not influence the cost of analysis. We are communicating with the JPF team to try to diagnose this performance issue with JPF. Even with the problematic optimization turned off, the na"ïve version still outperforms the intermediate language version on this large model. As we look deeper into the specifics of JPF, we also hope to see why the large model is not seeing the impressive performance improvements seen in the smaller ones.

5. PLEXIL LANGUAGE ISSUES

While analyzing and generating tests for PLEXIL plans, we discovered a problem in the Extended PLEXIL construct for If Then Else branches. In the final core PLEXIL output, the true and false branches became separate List nodes, with StartConditions dependent on the result of the condition. The Then branch only executed when the result was True, and the Else branch only ran if the result was False. The Unknown case was not accounted for, however, and so neither branch would start and the entire If Then Else construct was deadlocked. The Extended PLEXIL compiler was revised to run the Else branch ran if the condition was “not True”.

In addition, while developing a proof for an optimization of Sequence nodes (described more in Section 6), we discovered a problem in the core PLEXIL semantics. The optimization involves combining a Sequence of nodes into a single state machine, but in working through a proof sketch by hand it became obvious that the technique would only work under certain conditions. There are ways to defeat the sequential nature of the Sequence node and cause two nodes to run in parallel. For example, if a node’s SkipCondition becomes true, it can move to Finished and cause the next node to start, since its predecessor has Finished.

To better understand what properties would always hold for Sequence nodes and what conditions would cause the proof to be unsound, we constructed some NuSMV modules representing the original nodes and an optimized, combined version, and checked a property that said that our state mapping for the optimized version always matched the original. All the PLEXIL conditions were left to NuSMV to decide non-deterministically. When NuSMV came back with a counterexample, we discovered why the optimization was different and added a precondition that eliminated that possibility. The SkipCondition problem was discovered this way, as well as a few other properties that had to hold. However, at one point NuSMV produced a counterexample that we could not eliminate by introducing additional constraints. Normally, PLEXIL failures have a cascading effect, where a failing node causes its children to react to the failure either by finishing early or moving to a Failing state to wait for something to happen before finishing. However, the children of a Sequence node that experienced a failure could continue to execute normally if they were making the transition from Inactive to Waiting when the failure occurred. When the Sequence node recovered from its failure, the first node in the sequence could restart, even if another child was currently running, violating the requirement that only one node could ever be running at a time.

This particular series of events was more than just a problem for our optimization. When we constructed a PLEXIL plan that exhibited the same behavior and ran it in the reference PLEXIL executable, the execution crashed. The PLEXIL executable does not evaluate every condition every micro step, only the ones that could be used by the node or its children. This test case created a situation where a Finished node had an Executing node as a child, which the PLEXIL language designers did not think could happen. As a result, the ExitCondition of the parent was not ready when the children expected it. We sent the test case to the PLEXIL team, and they agreed that it was a problem that should be fixed in the PLEXIL semantics. In the meantime, we were able to quickly and easily edit our implementation of the semantics to verify that if the failure condition checks were added, our property would hold.

6. NEXT STEPS

We are currently working to understand the unusual results of the “Fluid” model. This involves investigating the interaction between the generated Java code, SPF and its underlying SAT-based constraint solver to see why our optimizations do not have the desired effect in the larger model.

The next step is more aggressive optimizations of the translated code. The Extended PLEXIL Sequence node automatically adds StartConditions to its children that ensure that each child node waits for its predecessor to enter the Finished state. This allows a PLEXIL programmer to easily avoid the parallel execution of nodes which occurs by default. It is also so commonly used that a keyword is required to avoid having nodes run in sequence. Also, many of the other Extended PLEXIL constructs are variations on a Sequence, such as the Try node, which executes in sequence until a node succeeds, then skips the rest. By definition, only one node in the list of child nodes will be performing any action, the rest will be Waiting for their turn to execute, or already Finished. However, in the step relation the nodes are all being checked for transitions in parallel. By merging the individual state machines into a single composite state machine, we can eliminate this unnecessary parallelism and represent all the nodes in the Sequence in a single state machine.

The merging process creates a composite state machine that links the end of one node state machine with the be-
Figure 9: Combined state machine for the Sequence node merging optimization.

The tags for each State contain the set of original PLEXIL states corresponding to the combined state. For example, consider the Executing state of the third node in a sequence of five. Each state in the combined state machine will have a tag for each of the five nodes. In this particular state, since the third node is tagged as Executing, the first and second nodes must have Finished, and the last two nodes must still be Waiting. Since PLEXIL nodes can refer to the state of other nodes in expressions, it is important that the “native” PLEXIL state is retrievable.

This optimization is obviously sound under certain node configurations. For example, at any time an ancestor node may fail and cause all the Waiting nodes to immediately finish with an outcome of Skipped. Without certain preconditions, it is also possible to have multiple nodes running in parallel. We did some preliminary analysis using NuSMV using arbitrary compositions of 3 and 4 PLEXIL nodes to determine sufficient conditions on the structure of the nodes to ensure the equivalence of the composition.

As described in Section 5, during this analysis we discovered that bugs in the PLEXIL semantics would have prevented the optimization from being correct. However, with those issues addressed, we found only a few preconditions that had to hold for the optimization to be correct: the Sequence node containing the sequence must retain its default ExitCondition, which keeps the node from finishing until all its children are Finished; otherwise it could restart and cause the first node to begin again even if the last node has not run yet. Also, the nodes in the sequence must also have their default SkipConditions and ExitConditions, otherwise the next node could start as described earlier.

To deal with the case where an ancestor’s failure causes all Waiting nodes to be skipped, the tags for those states have to be overridden. In NuSMV, we have proved that it is always true that a node with an outcome of Skipped must also be in the Finished state. When mapping to the native PLEXIL state of a node, we can first check the outcome of that node and return Finished, even if the current state is technically tagged as Waiting. In NuSMV, we have demonstrated the soundness of this merge process for up to four nodes, and created a sketch for a general proof of correctness for any number of nodes. We expect to complete a proof of the general case in the near future.

Now that we can generate test cases for PLEXIL plans, we must see how effective they are. Currently, we do not have test coverage information for the test cases that we are generating, and the specific test coverage criterion that is most effective for planning systems such as PLEXIL is something we plan to investigate. Ultimately, we would like to develop a tool that takes a PLEXIL plan and generates tests that reach a specified coverage goal, and perform analysis to confirm that a test case that has sufficient coverage can also provide assurance that a plan correctly achieves its goals.

7. CONCLUSION

Autonomous systems are critical to the success of future NASA missions, where latency and bandwidth constraints prohibit “tight loop” ground control. However, verification and validation of planning systems is not well explored, and there are specific challenges not found in most critical software. Towards this end, we have created tools that allow rigorous analysis of plans written in the PLEXIL notation, through a translator from PLEXIL into Java and the use of Symbolic Pathfinder. The execution model of PLEXIL — large numbers of state machines executing in parallel — makes efficient analysis difficult. We have recently added an intermediate language that supports merging of state machines and includes a variety of other analyses.

With our translation framework, we are able to generate rigorous test suites for a variety of models. In the process of generating tests and constructing our analysis tools, we have found two bugs in the PLEXIL language definition, one in the Extended PLEXIL macros and one in the formal semantics of the PLEXIL core language. These issues have been confirmed by the PLEXIL development team and corrected based on our feedback.

In the future, we believe that we can significantly improve the performance of our test generator by performing node-merging and other optimizations during translation, and by using alternate test generation metrics, such as MCDC, rather than path coverage.

8. REFERENCES