NASA/CR-2007-214542
NIA Report No. 2007-02

Model Checking Abstract PLEXIL Programs with SMART

Radu I. Siminiceanu
National Institute of Aerospace (NIA), Hampton, Virginia

April 2007

NASA/CR-2007-214542
NIA Report No. 2007-02

NATIONAL
INSTITUTE OF
AEROSPACE

Model Checking Abstract PLEXIL Programs with SMART

Radu I. Siminiceanu
National Institute of Aerospace (NIA), Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Cooperative Agreement NCC1-02043

April 2007

MODEL CHECKING ABSTRACT PLEXIL PROGRAMS WITH SMART

Radu I. Siminiceanu*

ABSTRACT

We describe a method to automatically generate discrete-state models of abstract Plan Execution Interchange Language (PLEXIL) programs that can be analyzed using model checking tools. Starting from a high-level description of a PLEXIL program or a family of programs with common characteristics, the generator lays the framework that models the principles of program execution. The concrete parts of the program are not automatically generated, but require the modeler to introduce them by hand. As a case study, we generate models to verify properties of the PLEXIL macro constructs that are introduced as shorthand notation. After an exhaustive analysis, we conclude that the macro definitions obey the intended semantics and behave as expected, but contingently on a few specific requirements on the timing semantics of micro-steps in the concrete executive implementation.

1 ABSTRACT MODELING OF PLEXIL PROGRAMS

This study is intended to investigate properties of general PLEXIL programs. The test suite for the experiment are standard program templates introduced as shorthand notation (frequently called "syntactic sugar") for PLEXIL [4] (section 5.2). The state transitions diagrams in [4] are used to derive a model of the execution of abstract programs in the tool SMART [2]. The modeling formalism of the tool is Stochastic Petri Nets [5]. A full description of SMART can be found in the user manual [3] (available on-line at `http://cs.ucr.edu/~ciardo/SMART/`).

Each node in a PLEXIL program can be modeled as an abstract automaton that specifies:

- the node type: `list, command, assign, function_call`;
- the node internal state: `inactive, waiting, executing, failing, iteration_ended, finishing, finished`, etc;
- six boolean variables representing the values of the following conditions (predicates): `start_condition, stop_condition, invariant_condition, pre_condition, post_condition, repeat_until_condition`;
- a variable to store the nature of the outcome: `success, failure, skipped, parent_failure`.

*National Institute of Aerospace, 100 Exploration Way, Hampton VA, 23666. E-mail: `radu@nianet.org`
This work was supported in part by the National Aeronautics and Space Administration under the cooperative agreement NCC-1-02043.

No other variables are considered at this point (local, global, program counters, etc.), but they can be manually introduced in the generated models as desired.

If a condition is defined as fully abstract, then its value can change independently, at any point in time, from `false` to `true` or vice versa. Such changes will trigger either internal transitions to a node (such as a start condition becoming true causing a node to transition from `waiting` to `executing` state), or global transitions (such as the invariant condition of a list node becoming `false` that triggers the failure of all its descendant nodes).

The automaton for the *local* transformations of a single node is the same for each category of nodes: list, command, assign, or function_call. The *global* transitions will reflect the interactions between these components, depending on the topology of the system.

1.1 Model builder input syntax

The model builder, written in C, builds the SMART automaton for the entire PLEXIL program with a simple compositional approach, from the description of the topology:

- The number of nodes $N > 0$ (on the first input file line);
- For each node, a description of:
  - node type: LIST, COMMAND, ASSIGN, FNCALL;
  - the parent-child relationships
  - condition types: abstract, concrete, or default.

  There is one node descriptor per line, with the syntax:

`NODE` *id node_type* `[CHILDREN` *id_list*`] [CONDITIONS [ABSTRACT|CONCRETE|DEFAULT]` *cond_list*`]`

Whenever a condition is not specifically defined, it is assumed to have a default value (*true* for all, except *end_condition* which is defined as "all children nodes finished")

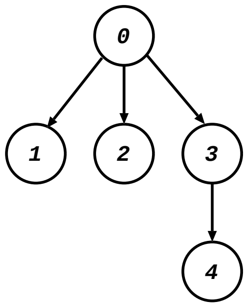

Figure 1: *Sample topology*

The example in Figure 1 could be described with the following input:

```
NODE 0 LIST CHILDREN 1 2 3 CONDITIONS ABSTRACT start end pre post until invariant
NODE 1 FNCALL CONDITIONS ABSTRACT start CONCRETE end
NODE 2 COMMAND CONDITIONS ABSTRACT start end pre post until invariant
NODE 3 LIST CHILDREN 4 CONDITIONS CONCRETE pre
NODE 4 ASSIGN CONCRETE
```

This input file represents an abstract program consisting of five nodes, where node 0 is a fully abstract list node with three child nodes $(1, 2, 3)$. Node 1 is a function call node with an abstract start condition and concrete end condition (to be manually introduced in the model), with the remaining four conditions taking the default value. Node 2 is a fully abstract command node. Node 3 is a list node with default conditions, except a concrete precondition, and a single child, node 4 (see Figure 1).

Using this approach, we instantiate the SMART model of the corresponding system topology (where a node in the figure is replaced by the appropriate automaton), and then investigate properties of interest related to the general execution flow of **all** PLEXIL programs represented by that abstract model.

This approach can be extended in an incremental manner, most importantly by allowing *concrete* conditions, but also by incorporating other program details, as desired. Also, this method should be orthogonal to the formal semantics chosen for PLEXIL, as the rules to execute the transitions can be implemented to reflect the semantic rules. At this stage, the models follow the PLEXIL document [4].

State space reduction.

In many circumstances (as it is the case of the macros) the majority of the node conditions have default values and remain unchanged throughout the entire execution of the program. The model builder has been optimized to prune entire sections of the model that are guaranteed to be disabled. For example, with a default invariant condition, all transitions dealing with a false invariant for that node, and also all the transitions dealing with a false ancestor invariant for its descendants can be eliminated. This helps in significantly reducing not only the state-space size, but also the size of the input file itself. An example of a pruned model is given in Appendix A.

Challenges.

A major issue in deciding a desired semantics for PLEXIL programs was the synchronous execution of micro-steps for quiescence. Since the number of micro-steps executed by a node in a quiescence cycle is not known in advance and, moreover, each node could execute a different number of steps, capturing the synchronous composition of this kind of automata, in either SAL [1] or SMART for abstract programs is not only a non-trivial task but it might have no representation in the two tools that we considered.

However, at the level of abstraction employed in this study, this issue has not had a critical impact on the analysis.

2 CASE STUDY: MACRO TEMPLATES

We next look at the following five macro templates defined in [4].

- Sequence;
- If-Then-Else, with the special case If-Then;
- While loop;
- For loop;

To interpret the topology figures, the solid lines represent parent–child relationships, dashed lines represent a consequential relationship enforced by setting the start condition of the target node to the condition that the source node is finished. Other relevant labels were added to the figure for the sake of clarity.

2.1 Sequence

Syntactic sugar for: `sequence N1, N2, ..., Nk`

```
Node:{
  NodeID: sequenceN1Nk;
  NodeList:{
    Node:{
      NodeID: doN1;
      NodeList:{ N1 }
    }
    Node:{
      NodeID: doN2;
      StartCondition: doN1.state == FINISHED;
      NodeList:{ N2 }
    }
    Node:{
      NodeID: doNk;
      StartCondition: doNk-1.state == FINISHED;
      NodeList:{ Nk }
    }
  }
}
```

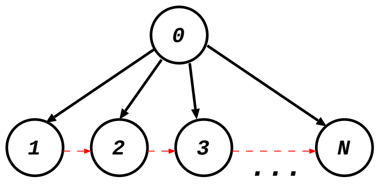

Figure 2: *Sequence macro topology*

An input file for the model builder is:

```
4
NODE 0 LIST CHILDREN 1 2 3
NODE 1 ASSIGN
NODE 2 ASSIGN CONDITIONS CONCRETE start
NODE 3 COMMAND CONDITIONS CONCRETE start
```

Discussion.

- The execution of the model produces a single final state, where each node has the outcome `SUCCESS`. This is the correct behavior.
- Moreover, the time intervals when the nodes N_i are in state `executing` do not overlap.

2.2 If-Then-Else

Macro definition: `if C then N1 else N2`

```
0: Node:{
     Boolean: which;
     NodeList:{
1:     Node:{
         NodeID: setup;
         Assignment: which = C;
       }
2:     Node:{
         NodeID: doIf;
         StartCondition: setup.state == FINISHED;
         EndCondition:
           isTrueNode.state == FINISHED ||
           isFalseNode.state == FINISHED
         NodeList:{
3:         Node:{
             NodeID: isTrueNode;
             StartCondition: which=true
             NodeList:{ N1 }
           }
4:         Node:{
             NodeID: isFalseNode;
             StartCondition: which=false
             NodeList:{ N2 }
           }
         }
       }
     }
   }
```

The input file for the model builder is:

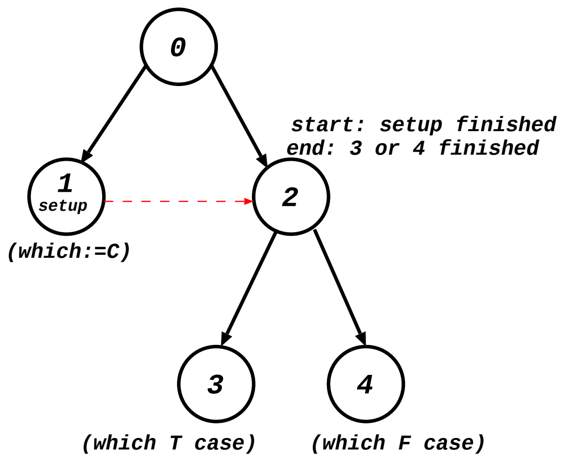

Figure 3: *If-then-else macro topology*

```
5
NODE 0 LIST CHILDREN 1 2
NODE 1 ASSIGN
NODE 2 LIST CHILDREN 3 4 CONDITIONS CONCRETE start end
NODE 3 COMMAND CONDITIONS ABSTRACT start
NODE 4 COMMAND CONDITIONS CONCRETE start
```

Discussion.

- The execution of the model ends in two possible states:
  - When the value of the variable *which* is *true*, node 4 is skipped and node 3 is executed;
  - When the value of the variable *which* is *false*, node 3 is skipped and node 4 is executed;

  This should be the desired behavior. However,

- This is contingent on the value of *which* being set to C at the correct moment in time, that is when the macro execution starts. For the macro, the value of C is read at a time corresponding to node 0 being executed. In the expanded program, *which* is set by node 2.

∗ Therefore, we derive the following requirement on the execution model:
"The total time observed between the moments when the execution of node 0 and node 2 starts is 0*"*
or
"The value of C does not change between the moments when the execution of node 0 and node 2 starts".

2.2.1 Special case: If-Then

Macro definition: `if C then N`

```
0: Node:{
     Boolean: which;
     NodeList:{
1:     Node:{
         NodeID: setup;
         Assignment: which = C;
       }
2:     Node:{
         NodeID: doIf;
         StartCondition: setup.state == FINISHED;
         EndCondition:
           which == false |
           isTrueNode.state == FINISHED
         NodeList:{
3:         Node:{
             NodeID: isTrueNode;
             StartCondition: which=true
             NodeList:{ N }
           }
         }
       }
     }
   }
```

The input file for the model builder is:

```
4
NODE 0 LIST CHILDREN 1 2
NODE 1 ASSIGN
NODE 2 LIST CHILDREN 3 4 CONDITIONS CONCRETE start end
NODE 3 COMMAND CONDITIONS ABSTRACT start
```

Discussion.

- Similar to the general case, the execution of the model ends in two possible states:
  - When the value of the variable *which* is *true*, node 3 is executed;
  - When the value of the variable *which* is *false*, node 3 is skipped;

  This is the desired behavior, contingent on the same prerequisites as in the general case.

2.3 While loop

Macro definition: `while C do N`

```
0: **Node:{**
     **Boolean:** which;
     **NodeList:{**
1:     **Node:{**
         **NodeID:** setup;
         **Assignment:** which = C;
       **}**
2:     **Node:{**
         **NodeID:** doWhile;
         **StartCondition:** setup.state == FINISHED;
         **EndCondition:**
           which == false |
           isTrueNode.state == FINISHED
         **NodeList:{**
3:         **Node:{**
             **NodeID:** isTrueNode;
             **StartCondition:** which=true
             **Repeat-until-condition:** not C;
4:           **NodeList:{** N **}**
           **}**
         **}**
       **}**
     **}**
   **}**
```

The input file for the model builder is:

```
5
NODE 0 LIST CHILDREN 1 2
NODE 1 ASSIGN
NODE 2 LIST CHILDREN 3 CONDITIONS CONCRETE start end
NODE 3 LIST CHILDREN 4 CONDITIONS CONCRETE start until
NODE 4 COMMAND
```

Discussion.

- Intuitively, one would expect that the execution always ends in a state where C is false, capturing the last (failed) attempt to execute the loop. In reality, the analysis is not as straightforward. Nevertheless, the behavior of the macro can be deemed as correct.

  In our model, the execution of the macro ends in two types of states:

  1. Nodes 3 and 4 are skipped, the value of variable *which* is false;

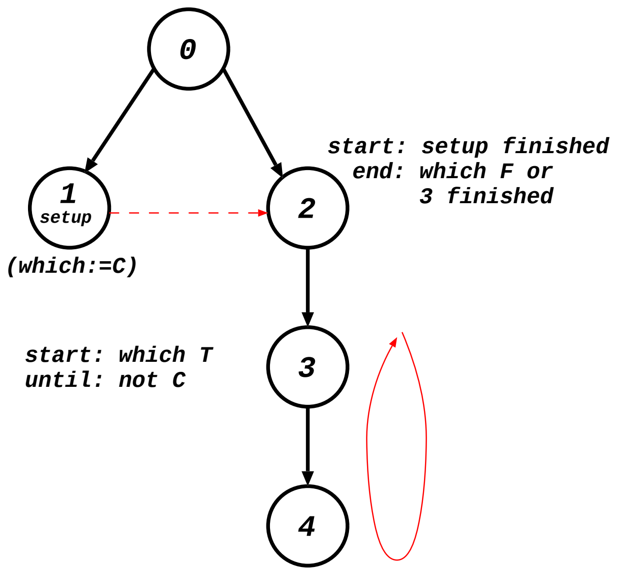

Figure 4: *While loop macro topology*

2. Nodes 3 and 4 finish normally after executing at least once, the value of variable *which* is true.

In both cases, the value of C in the final states does not necessarily match that of *which*. This situation occurs when C changes after the loop execution ends. Note that C is an abstract expression, so its value should be independent of other model variables and has no restrictions on whether and when it changes.

- Additionally, it can be checked that the value of variable *which* is always true when node 4 executes; Note that the same cannot be stated of the value of C;
- A requirement on the timing of storing the value of C to *which* similar to the If-Then-Else case is again needed;

2.4 For loops

Macro definition: `for (int counter=init(Z), C(counter,X), f(counter,Y)) N`

```
0: Node:{
     Integer: counter;
     Boolean: which;
     NodeList:{
1:     Node:{
         NodeID: setup;
         Assignment: counter = init(Z);
       }
2:     Node:{
         NodeID: setup;
```

```
          Assignment: which = C(counter,X);
        }
3:      Node:{
          NodeID: doWhile;
          StartCondition: setup.state == FINISHED;
          EndCondition:
            which == false |
            doLoop.state == FINISHED
          NodeList:{
4:          Node:{
              NodeID: doLoop;
              Repeat-until-condition: ! C(counter, X);
              NodeList:{
5:              Node:{
                  NodeId: doN;
                  NodeList:{ N }
                }
6:              Node:{
                  NodeId: counterUpdate;
                  StartCondition: doN.state == FINISHED;
                  Assignment: counter = f(counter,Y);
                }
              }
            }
          }
        }
      }
    }
```

The input file for the model builder is:

```
7
NODE 0 LIST CHILDREN 1 2 3
NODE 1 ASSIGN
NODE 2 ASSIGN
NODE 3 LIST CHILDREN 4 CONDITIONS CONCRETE start end
NODE 4 LIST CHILDREN 5 6 CONDITIONS CONCRETE until
NODE 5 COMMAND
NODE 6 ASSIGN CONDITIONS CONCRETE start
```

Discussion.

The execution of this macro is significantly more complex than the previous ones. We have several observations that might present some concern.

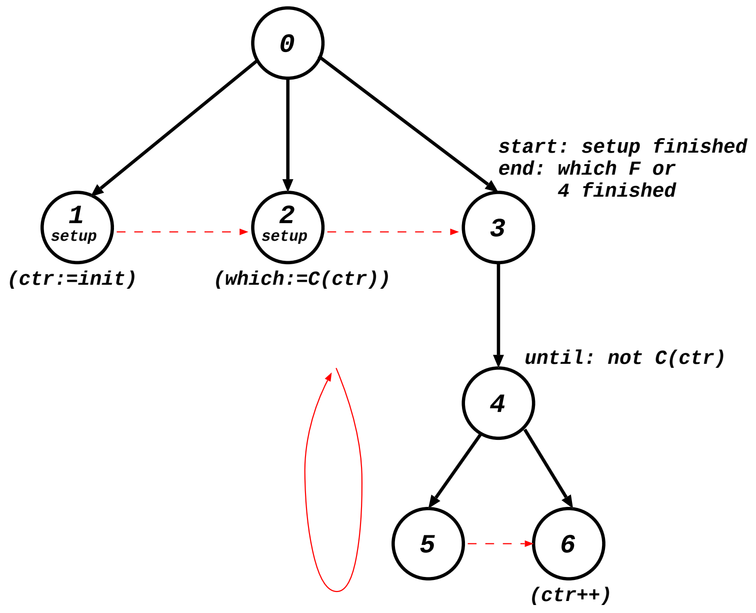

Figure 5: *For loop macro topology*

- It is possible that the for loop starts with an *uninitialized* counter.

  This scenario unfolds as follows:

  - Node 0 starts executing;
  - Nodes 1 and 2 have default start condition, hence they will simultaneously start executing, too.
  - Node 1 sets the value of *counter* to the initial value $init(Z)$; *concurrently*, node 2 decides the value of *which* based on the uninitialized value of *counter*, $which = C(counter, X)$.
  - The for loop is then executed or not, based on a nondeterministic choice (whether the uninitialized value of *counter* was indeed satisfying the condition C or not), rather than a deterministic one;

  This situation can be fixed by requiring that node 2 starts only after node 1 has finished, and thus reads the correct value of *counter*;

- It should be expected that the execution ends with the first value of the counter that does not satisfy the finish criteria $f(counter, Y)$. This does not happen in our model. To make the analysis easier to understand, we illustrate the faulty scenario by means of the traditional case of a simple counter increment

  ```
  for (int counter = min, counter++, counter < max) N;
  ```

  instead of the general format.

- When $min < max$, the programs yields the expected results: node N is executed exactly $max - min$ times, and ends with $counter = max$.
- However, when $min \geq max$, the model might not have only the expected outcome (that is, node N is skipped) but could also exhibit one behavior when node N is actually executed once. This is contingent on the interpretation of priorities, as explained below. The scenario unfolds as follows:
  - Node 2 sets the value of *which* to false;
  - Node 3 starts executing, also triggering the move of its child node 4 (*doLoop*) from `inactive` to `waiting`;
  - Node 3 immediately (i.e. in the next micro-step) satisfies its end condition (the term $which == false$);
  - Simultaneously, node 4 has two racing events: *ancestor ends* and *start condition* (default) true; If the **higher priority** is given to the start condition (priority 3 vs. 2), node 4 proceeds to executing one iteration;
  - Eventually, upon entering state `iteration_ended`, node 4 does not repeat (because $counter \geq max$) and the program finally ends.
- The situation can be corrected by adding a start condition to node 4 to rule out the possibility of executing when *which* is false.

2.5 Adding a fully abstract context

The models studied so far have been stand-alone, where the root node of the macro is also the root node of the entire PLEXIL program and all variables are either concrete or default. While the study offers guarantees that the macros behave as expected in isolation, in order to make more general statements, they have to be studied in a *context*. To this extent, we can add a fully abstract node as parent of the root node of the macro model. Since its conditions are fully independent, the abstract node serves the purpose of *exhaustively* checking the behavior of the macro in all possible contexts.
For example, the model of the If-Then-Else macro in a context is described by the following input:

```
6
NODE 5 LIST CHILDREN 0 CONDITIONS ABSTRACT start end pre post until invariant
NODE 0 LIST CHILDREN 1 2
NODE 1 ASSIGN
NODE 2 LIST CHILDREN 3 4 CONDITIONS CONCRETE start end
NODE 3 COMMAND CONDITIONS ABSTRACT start
NODE 4 COMMAND CONDITIONS CONCRETE start
```

As expected, the major challenge is the state-space explosion, mostly due to the large number of possible interleavings of external events (that change the value of the abstract conditions). It can be equally attributed to a number of state transitions that are *only* enabled in the general context, such as those caused by invariant failure or parent failure, that were not present in the stand-alone macro.

As a measure of this state-space increase (NB: due to a ***single*** abstract node!), the number of states in the If-Then-Else macro with context balloons from 80 to $3,267,980,384$. These kind of state-spaces can usually be handled only by symbolic model checkers.

3 CONCLUSIONS. FUTURE WORK.

We described a method to automatically generate discrete-state models of abstract PLEXIL programs that can be analyzed using model checking tools. We generated models of the PLEXIL macro constructs that are introduced as syntactic sugar to the language and exhaustively checked the reachable states in the models to verify that they obey the intended semantics. This technique is of general purpose and can be used to model any concrete PLEXIL program and investigate its properties, granted that the symbolic state-space of the model fits into the maximum 4GB of memory on a 32-bit platform.

A similar model builder is currently being developed to generate output models for the tool SAL [1]. The intent is to utilize the wider array of model checking techniques and the more expressive input language to be able to analyze other aspects of interest, beside the basic safety properties.

REFERENCES

[1] L. de Moura, S. Owre, and N. Shankar. The SAL Language Manual. Technical report, SRI International, Menlo Park, CA, USA, Aug. 2003.

[2] Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu. Logical and stochastic modeling with SMART. In *Modeling Techniques and Tools for Computer Perf. Eval.*, LNCS 2794, pages 78–97, Urbana, IL, USA, September 2003. Springer-Verlag.

[3] Gianfranco Ciardo, Andrew Miner, and Radu Siminiceanu. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. http://cs.ucr.edu/∼ciardo/SMART/.

[4] Tara Estlin, Ari Jónsson, Corina Păsăreanu, Reid Simmons, Kam Tso, and Vandi Verma. Plan execution interchange language (PLEXIL). Technical report, NASA Ames, September 2005.

[5] Tadao Murata. Petri Nets: properties, analysis and applications. *Proc. IEEE*, 77(4):541–579, April 1989.

APPENDIX A

The SMART model of the if-then-else macro template is listed below:

```
/*****************************************
   Model generated by SMB v3.0 (July 2006)
   Author: Radu I. Siminiceanu (NIA)
*****************************************/


// Topology

// Node 0: type LIST parent: -1 children: 1 2
// Node 1: type ASSIGN parent: 0 children:
// Node 2: type LIST parent: 0 children: 3 4
// Node 3: type COMMAND parent: 2 children:
// Node 4: type COMMAND parent: 2 children:


spn plexil := {
    place
      st_inactive_0,
      st_waiting_0,
      st_executing_0,
      st_failing_0,
      st_iter_ended_0,
      st_finishing_0,
      st_finished_0,

      ancestor_end_0,
      ancestor_failed_0,

      outcome_success_0,
      outcome_failure_0,
      outcome_parent_failure_0,
      outcome_skipped_0;

    partition(
      st_inactive_0:st_waiting_0:st_executing_0:
      st_finishing_0:st_failing_0:st_iter_ended_0:st_finished_0:
      ancestor_end_0:ancestor_failed_0:
      outcome_success_0:outcome_failure_0:outcome_parent_failure_0:
      outcome_skipped_0
    );
    place
      st_inactive_1,
      st_waiting_1,
      st_executing_1,
      st_failing_1,
      st_iter_ended_1,
      st_finishing_1,
      st_finished_1,

      ancestor_end_1,
      ancestor_failed_1,

/* MANUALLY ADDED */ which, C,

      outcome_success_1,
      outcome_failure_1,
      outcome_parent_failure_1,
      outcome_skipped_1;

    partition(
      which:C,
      st_inactive_1:st_waiting_1:st_executing_1:
      st_finishing_1:st_failing_1:st_iter_ended_1:st_finished_1:
      ancestor_end_1:ancestor_failed_1:
      outcome_success_1:outcome_failure_1:outcome_parent_failure_1:
      outcome_skipped_1
    );

    place
      st_inactive_2,
      st_waiting_2,
      st_executing_2,
      st_failing_2,
      st_iter_ended_2,
      st_finishing_2,
      st_finished_2,

      ancestor_end_2,
      ancestor_failed_2,

      outcome_success_2,
      outcome_failure_2,
      outcome_parent_failure_2,
      outcome_skipped_2;
```

```
partition(
  st_inactive_2:st_waiting_2:st_executing_2:
  st_finishing_2:st_failing_2:st_iter_ended_2:st_finished_2:
  ancestor_end_2:ancestor_failed_2:
  outcome_success_2:outcome_failure_2:outcome_parent_failure_2:
  outcome_skipped_2
);

place
  st_inactive_3,
  st_waiting_3,
  st_executing_3,
  st_failing_3,
  st_iter_ended_3,
  st_finishing_3,
  st_finished_3,

  ancestor_end_3,
  ancestor_failed_3,

  outcome_success_3,
  outcome_failure_3,
  outcome_parent_failure_3,
  outcome_skipped_3;

partition(
  st_inactive_3:st_waiting_3:st_executing_3:
  st_finishing_3:st_failing_3:st_iter_ended_3:st_finished_3:
  ancestor_end_3:ancestor_failed_3:
  outcome_success_3:outcome_failure_3:outcome_parent_failure_3:
  outcome_skipped_3
);

place
  st_inactive_4,
  st_waiting_4,
  st_executing_4,
  st_failing_4,
  st_iter_ended_4,
  st_finishing_4,
  st_finished_4,

  ancestor_end_4,
  ancestor_failed_4,
      outcome_success_4,
      outcome_failure_4,
      outcome_parent_failure_4,
      outcome_skipped_4;

    partition(
      st_inactive_4:st_waiting_4:st_executing_4:
      st_finishing_4:st_failing_4:st_iter_ended_4:st_finished_4:
      ancestor_end_4:ancestor_failed_4:
      outcome_success_4:outcome_failure_4:outcome_parent_failure_4:
      outcome_skipped_4
    );


//----------------------------------------
//  Local transitions for list node #0
//----------------------------------------

    trans
      t_inactive_to_waiting_0;

    arcs(
      st_inactive_0:t_inactive_to_waiting_0,
        t_inactive_to_waiting_0:st_waiting_0
    );

    trans
      t_waiting_to_executing_0;

    arcs(
      st_waiting_0:t_waiting_to_executing_0,
        t_waiting_to_executing_0:st_executing_0
    );

    trans
      t_executing_end_condition_true_post_condition_true_0;

    arcs(
      st_executing_0:t_executing_end_condition_true_post_condition_true_0,
        t_executing_end_condition_true_post_condition_true_0:st_finishing_0,
        outcome_success_0:t_executing_end_condition_true_post_condition_true_0:
          tk(outcome_success_0),
        t_executing_end_condition_true_post_condition_true_0:outcome_success_0
    );
    guard(
      t_executing_end_condition_true_post_condition_true_0:
/* MANUALLY ADDED */ tk(st_finished_1)>0 & tk(st_finished_2)>0
    );

    trans
      t_failing_to_finished_0;

    arcs(
      st_failing_0:t_failing_to_finished_0,
        t_failing_to_finished_0:st_finished_0
    );

    guard(
      t_failing_to_finished_0:
        tk(outcome_parent_failure_0)==1
    );

    trans
      t_finishing_post_true_0;

    arcs(
      st_finishing_0:t_finishing_post_true_0,
        t_finishing_post_true_0:st_iter_ended_0,
        outcome_success_0:t_finishing_post_true_0:tk(outcome_success_0),
        t_finishing_post_true_0:outcome_success_0
    );

    trans
      t_iter_ended_no_repeat_0;

    arcs(
      st_iter_ended_0:t_iter_ended_no_repeat_0,
        t_iter_ended_no_repeat_0:st_finished_0
    );

    init(st_inactive_0:1);

//-----------------------------------------
//  Local transitions for assign node #1
//-----------------------------------------

    trans
      t_inactive_ancestor_ends_1,
      t_inactive_to_waiting_1;
    arcs(
      st_inactive_1:t_inactive_ancestor_ends_1,
        t_inactive_ancestor_ends_1:st_finished_1,
        outcome_skipped_1:t_inactive_ancestor_ends_1:tk(outcome_skipped_1),
        t_inactive_ancestor_ends_1:outcome_skipped_1,
        ancestor_end_1:t_inactive_ancestor_ends_1,
      st_inactive_1:t_inactive_to_waiting_1,
        t_inactive_to_waiting_1:st_waiting_1
    );

    guard(
      t_inactive_to_waiting_1:tk(st_executing_0)>0,
      t_inactive_ancestor_ends_1:tk(st_executing_0)==0 & tk(ancestor_end_1)==1
    );

    trans
      t_waiting_ancestor_ends_1,
      t_waiting_to_executing_1;

    arcs(
      st_waiting_1:t_waiting_ancestor_ends_1,
        ancestor_end_1:t_waiting_ancestor_ends_1,
        t_waiting_ancestor_ends_1:st_finished_1,
        outcome_skipped_1:t_waiting_ancestor_ends_1:tk(outcome_skipped_1),
        t_waiting_ancestor_ends_1:outcome_skipped_1,
      st_waiting_1:t_waiting_to_executing_1,
/* MANUALLY ADDED */
        which:t_waiting_to_executing_1:tk(which),
        t_waiting_to_executing_1:which:tk(C),
        t_waiting_to_executing_1:st_executing_1
    );

    guard(
      t_waiting_ancestor_ends_1:
        tk(ancestor_end_1)==1
    );

    trans
      t_executing_end_condition_true_post_condition_true_1;

    arcs(
      st_executing_1:t_executing_end_condition_true_post_condition_true_1,
        t_executing_end_condition_true_post_condition_true_1:st_iter_ended_1,
        outcome_success_1:t_executing_end_condition_true_post_condition_true_1:
          tk(outcome_success_1),
        t_executing_end_condition_true_post_condition_true_1:outcome_success_1
    );

    trans
      t_iter_ended_no_repeat_1;

    arcs(
      st_iter_ended_1:t_iter_ended_no_repeat_1,
        t_iter_ended_no_repeat_1:st_finished_1
    );

    trans
      t_finished_to_inactive_1;

    arcs(
      st_finished_1:t_finished_to_inactive_1,
        t_finished_to_inactive_1:st_inactive_1,
        outcome_success_1:t_finished_to_inactive_1:tk(outcome_success_1),
        outcome_failure_1:t_finished_to_inactive_1:tk(outcome_failure_1),
        outcome_parent_failure_1:t_finished_to_inactive_1:
        tk(outcome_parent_failure_1)
    );

    guard(t_finished_to_inactive_1:tk(st_waiting_0)==1);

    init(st_inactive_1:1);

//-----------------------------------------
//  Local transitions for list node #2
//-----------------------------------------

    trans
      t_inactive_ancestor_ends_2,
      t_inactive_to_waiting_2;

    arcs(
      st_inactive_2:t_inactive_ancestor_ends_2,
        t_inactive_ancestor_ends_2:st_finished_2,
        outcome_skipped_2:t_inactive_ancestor_ends_2:tk(outcome_skipped_2),
        t_inactive_ancestor_ends_2:outcome_skipped_2,
        ancestor_end_2:t_inactive_ancestor_ends_2,
      st_inactive_2:t_inactive_to_waiting_2,
        t_inactive_to_waiting_2:st_waiting_2
    );
    guard(
      t_inactive_to_waiting_2:tk(st_executing_0)>0,
      t_inactive_ancestor_ends_2:tk(st_executing_0)==0 & tk(ancestor_end_2)==1
    );

    trans
      t_waiting_ancestor_ends_2,
      t_waiting_to_executing_2;

    arcs(
      st_waiting_2:t_waiting_ancestor_ends_2,
        ancestor_end_2:t_waiting_ancestor_ends_2,
        t_waiting_ancestor_ends_2:st_finished_2,
        outcome_skipped_2:t_waiting_ancestor_ends_2:tk(outcome_skipped_2),
        t_waiting_ancestor_ends_2:outcome_skipped_2,
      st_waiting_2:t_waiting_to_executing_2,
        t_waiting_to_executing_2:st_executing_2
    );

    guard(
      t_waiting_ancestor_ends_2:
        tk(ancestor_end_2)==1,
      t_waiting_to_executing_2:
/* MANUALLY MODIFIED */
        tk(st_finished_1)==1
    );

    trans
/* MANUALLY ADDED */ t_executing_end_condition_true_post_condition_true_2a,
      t_executing_end_condition_true_post_condition_true_2;

    arcs(
/* MANUALLY ADDED + MODIFIED */
      st_executing_2:t_executing_end_condition_true_post_condition_true_2a,
        t_executing_end_condition_true_post_condition_true_2a:st_finishing_2,
        outcome_success_2:t_executing_end_condition_true_post_condition_true_2a:
          tk(outcome_success_2),
        t_executing_end_condition_true_post_condition_true_2a:outcome_success_2,
        t_executing_end_condition_true_post_condition_true_2a:ancestor_end_3,
      st_executing_2:t_executing_end_condition_true_post_condition_true_2,
        t_executing_end_condition_true_post_condition_true_2:st_finishing_2,
        outcome_success_2:t_executing_end_condition_true_post_condition_true_2:
          tk(outcome_success_2),
        t_executing_end_condition_true_post_condition_true_2:ancestor_end_4,
        t_executing_end_condition_true_post_condition_true_2:outcome_success_2
    );

    guard(
      t_executing_end_condition_true_post_condition_true_2:
/* MANUALLY ADDED */ tk(st_finished_3)>0,
      t_executing_end_condition_true_post_condition_true_2a:
/* MANUALLY ADDED */ tk(st_finished_4)>0
    );

    trans
      t_failing_to_finished_2;

    arcs(
      st_failing_2:t_failing_to_finished_2,
        t_failing_to_finished_2:st_finished_2
    );

    guard(
      t_failing_to_finished_2:
        tk(outcome_parent_failure_2)==1
    );

    trans
      t_finishing_post_true_2;

    arcs(
      st_finishing_2:t_finishing_post_true_2,
        t_finishing_post_true_2:st_iter_ended_2,
        outcome_success_2:t_finishing_post_true_2:tk(outcome_success_2),
        t_finishing_post_true_2:outcome_success_2
    );

    trans
      t_iter_ended_no_repeat_2;

    arcs(
      st_iter_ended_2:t_iter_ended_no_repeat_2,
        t_iter_ended_no_repeat_2:st_finished_2
    );

    trans
      t_finished_to_inactive_2;

    arcs(
      st_finished_2:t_finished_to_inactive_2,
        t_finished_to_inactive_2:st_inactive_2,
        outcome_success_2:t_finished_to_inactive_2:tk(outcome_success_2),
        outcome_failure_2:t_finished_to_inactive_2:tk(outcome_failure_2),
        outcome_parent_failure_2:t_finished_to_inactive_2:
        tk(outcome_parent_failure_2)
    );

    guard(t_finished_to_inactive_2:tk(st_waiting_0)==1);

    init(st_inactive_2:1);

//----------------------------------------
//  Local transitions for command node #3
//----------------------------------------

    trans
      C_goes_false,
      C_goes_true;

    arcs(
      C:C_goes_false,
      C_goes_true:C
    );

    inhibit(
      C:C_goes_true
    );

    guard(
/* MANUALLY MODIFIED */
      C_goes_false:
        (tk(st_inactive_1)==1 | tk(st_waiting_1)==1),
      C_goes_true:
        (tk(st_inactive_1)==1 | tk(st_waiting_1)==1)
    );

    trans
      t_inactive_ancestor_ends_3,
      t_inactive_to_waiting_3;

    arcs(
      st_inactive_3:t_inactive_ancestor_ends_3,
        t_inactive_ancestor_ends_3:st_finished_3,
        outcome_skipped_3:t_inactive_ancestor_ends_3:tk(outcome_skipped_3),
        t_inactive_ancestor_ends_3:outcome_skipped_3,
        ancestor_end_3:t_inactive_ancestor_ends_3,
      st_inactive_3:t_inactive_to_waiting_3,
        t_inactive_to_waiting_3:st_waiting_3
    );

    guard(
      t_inactive_to_waiting_3:tk(st_executing_2)>0,
      t_inactive_ancestor_ends_3:tk(st_executing_2)==0 & tk(ancestor_end_3)==1
    );

    trans
      t_waiting_ancestor_ends_3,
      t_waiting_to_executing_3;

    arcs(
      st_waiting_3:t_waiting_ancestor_ends_3,
        ancestor_end_3:t_waiting_ancestor_ends_3,
        t_waiting_ancestor_ends_3:st_finished_3,
        outcome_skipped_3:t_waiting_ancestor_ends_3:tk(outcome_skipped_3),
        t_waiting_ancestor_ends_3:outcome_skipped_3,
      st_waiting_3:t_waiting_to_executing_3,
        t_waiting_to_executing_3:st_executing_3
    );

    guard(
/* MANUALLY MODIFIED */
      t_waiting_ancestor_ends_3:
        tk(which)==0 & tk(ancestor_end_3)==1,
      t_waiting_to_executing_3:
        tk(which)==1
    );

    trans
      t_executing_end_condition_true_post_condition_true_3;

    arcs(
      st_executing_3:t_executing_end_condition_true_post_condition_true_3,
        t_executing_end_condition_true_post_condition_true_3:st_iter_ended_3,
        outcome_success_3:t_executing_end_condition_true_post_condition_true_3:
          tk(outcome_success_3),
        t_executing_end_condition_true_post_condition_true_3:outcome_success_3
    );

    trans
```

```
  t_failing_abort_incomplete_3,
  t_failing_to_finished_3;

arcs(
  st_failing_3:t_failing_abort_incomplete_3,
    t_failing_abort_incomplete_3:st_failing_3,
  st_failing_3:t_failing_to_finished_3,
    t_failing_to_finished_3:st_finished_3
);

guard(
  t_failing_to_finished_3:
    tk(outcome_parent_failure_3)==1
);

trans
  t_finishing_post_true_3;

arcs(
  st_finishing_3:t_finishing_post_true_3,
    t_finishing_post_true_3:st_iter_ended_3,
    outcome_success_3:t_finishing_post_true_3:tk(outcome_success_3),
    t_finishing_post_true_3:outcome_success_3
);

trans
  t_iter_ended_no_repeat_3;

arcs(
  st_iter_ended_3:t_iter_ended_no_repeat_3,
    t_iter_ended_no_repeat_3:st_finished_3
);

trans
  t_finished_to_inactive_3;

arcs(
  st_finished_3:t_finished_to_inactive_3,
    t_finished_to_inactive_3:st_inactive_3,
    outcome_success_3:t_finished_to_inactive_3:tk(outcome_success_3),
    outcome_failure_3:t_finished_to_inactive_3:tk(outcome_failure_3),
    outcome_parent_failure_3:t_finished_to_inactive_3:
    tk(outcome_parent_failure_3)
);
```

```
    guard(t_finished_to_inactive_3:tk(st_waiting_2)==1);

    init(st_inactive_3:1);

//----------------------------------------
//  Local transitions for command node #4
//----------------------------------------

    trans
      t_inactive_ancestor_ends_4,
      t_inactive_to_waiting_4;

    arcs(
      st_inactive_4:t_inactive_ancestor_ends_4,
        t_inactive_ancestor_ends_4:st_finished_4,
        outcome_skipped_4:t_inactive_ancestor_ends_4:tk(outcome_skipped_4),
        t_inactive_ancestor_ends_4:outcome_skipped_4,
        ancestor_end_4:t_inactive_ancestor_ends_4,
      st_inactive_4:t_inactive_to_waiting_4,
        t_inactive_to_waiting_4:st_waiting_4
    );

    guard(
      t_inactive_to_waiting_4:tk(st_executing_2)>0,
      t_inactive_ancestor_ends_4:tk(st_executing_2)==0 & tk(ancestor_end_4)==1
    );

    trans
      t_waiting_ancestor_ends_4,
      t_waiting_to_executing_4;

    arcs(
      st_waiting_4:t_waiting_ancestor_ends_4,
        ancestor_end_4:t_waiting_ancestor_ends_4,
        t_waiting_ancestor_ends_4:st_finished_4,
        outcome_skipped_4:t_waiting_ancestor_ends_4:tk(outcome_skipped_4),
        t_waiting_ancestor_ends_4:outcome_skipped_4,
      st_waiting_4:t_waiting_to_executing_4,
        t_waiting_to_executing_4:st_executing_4
    );

    guard(
      t_waiting_ancestor_ends_4:
        tk(which)==1 & tk(ancestor_end_4)==1,
      t_waiting_to_executing_4:
```

```
/* MANUALLY MODIFIED */
        tk(which)==0
    );

    trans
      t_executing_end_condition_true_post_condition_true_4;

    arcs(
      st_executing_4:t_executing_end_condition_true_post_condition_true_4,
        t_executing_end_condition_true_post_condition_true_4:st_iter_ended_4,
        outcome_success_4:t_executing_end_condition_true_post_condition_true_4:
          tk(outcome_success_4),
        t_executing_end_condition_true_post_condition_true_4:outcome_success_4
    );

    trans
      t_failing_abort_incomplete_4,
      t_failing_to_finished_4;

    arcs(
      st_failing_4:t_failing_abort_incomplete_4,
        t_failing_abort_incomplete_4:st_failing_4,
      st_failing_4:t_failing_to_finished_4,
        t_failing_to_finished_4:st_finished_4
    );

    guard(
      t_failing_to_finished_4:
        tk(outcome_parent_failure_4)==1
    );

    trans
      t_finishing_post_true_4;

    arcs(
      st_finishing_4:t_finishing_post_true_4,
        t_finishing_post_true_4:st_iter_ended_4,
        outcome_success_4:t_finishing_post_true_4:tk(outcome_success_4),
        t_finishing_post_true_4:outcome_success_4
    );

    trans
      t_iter_ended_no_repeat_4;

    arcs(
      st_iter_ended_4:t_iter_ended_no_repeat_4,
        t_iter_ended_no_repeat_4:st_finished_4
    );

    trans
      t_finished_to_inactive_4;

    arcs(
      st_finished_4:t_finished_to_inactive_4,
        t_finished_to_inactive_4:st_inactive_4,
        outcome_success_4:t_finished_to_inactive_4:tk(outcome_success_4),
        outcome_failure_4:t_finished_to_inactive_4:tk(outcome_failure_4),
        outcome_parent_failure_4:t_finished_to_inactive_4:
        tk(outcome_parent_failure_4)
    );

    guard(t_finished_to_inactive_4:tk(st_waiting_2)==1);

    init(st_inactive_4:1);

//-----------------
//  Queries
//-----------------

  bool db   := debug;
  bigint ns := num_states(false);

  stateset fin := difference(reachable, prev(potential(true)));
  bigint nfin  := card(fin);
  bool prn_fin := printset(fin);
};

#StateStorage MDD_SATURATION
#Verbose true

compute(plexil.db);
print("Number of reachable states: ", plexil.ns, "\n");
print("\nExecution ends in the following states:\n");
compute(plexil.prn_fin);
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01/04/2007	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Model Checking Abstract PLEXIL Programs with SMART	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER NCC1-02043
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Radu I. Siminiceanu	**5d. PROJECT NUMBER** 2605
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center, Hampton, VA 23681-2199; National Institute of Aerospace (NIA), 100 Exploration Way, Hampton, VA 23666	**8. PERFORMING ORGANIZATION REPORT NUMBER** NIA Report No. 2007-02
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, Washington, DC 20546-0001	**10. SPONSORING/MONITOR'S ACRONYM(S)** NASA
	11. SPONSORING/MONITORING REPORT NUMBER NASA/CR-2007-214542

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES

Langley Technical Monitor: Ben Di Vito

14. ABSTRACT

We describe a method to automatically generate discrete-state models of abstract Plan Execution Interchange Language (PLEXIL) programs that can be analyzed using model checking tools. Starting from a high-level description of a PLEXIL program or a family of programs with common characteristics, the generator lays the framework that models the principles of program execution. The concrete parts of the program are not automatically generated, but require the modeler to introduce them by hand. As a case study, we generate models to verify properties of the PLEXIL macro constructs that are introduced as shorthand notation. After an exhaustive analysis, we conclude that the macro definitions obey the intended semantics and behave as expected, but contingently on a few specific requirements on the timing semantics of micro-steps in the concrete executive implementation.

15. SUBJECT TERMS

Model Checking; Planning; Spacecraft Autonomy

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19b. NAME OF RESPONSIBLE PERSON** STI Help Desk (email: help@sti.nasa.gov)
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	33	**19b. TELEPHONE NUMBER** ***(Include area code)*** (301) 621-0390