# Mother2 and MOSS: Automated Test Generation from Real-Time Requirements

*Joe Maybee*

*Tektronix, Inc.*
*Graphic Printing and Imaging Division*
*P.O. Box 1000, M.S. 63-430*
*Wilsonville, Or 97070*
*maybee@pogo.wv.tek.com*

*Portland State University*
*Department of Computer Science*
*P.O. Box 751*
*Portland, Or 97207*
*maybee@cs.pdx.edu*

## Abstract

Can the effort used for test writing be used to write requirements instead? A simple requirements specification system combined with an automated test fixture holds great promise. This paper examines one such approach.

*Keywords and Phrases:* Prolog, real-time embedded systems, formal methods, automated test methods, requirements testing, electromechanical systems.

*Biographical:* Joe Maybee has been building real-time embedded systems for Tektronix, Inc. since 1978. Recently, Joe has focused on software quality in reactive systems and the accurate definition of software requirements. Joe also teaches Software Engineering at Portland State University.

Joe is the recipient of an *Outstanding Achievement Award for Innovation* from the *1992 Pacific Northwest Software Quality Conference* for his work with the MOTHER test system, and first-place winner at the *1993 Tektronix Engineering Conference* for his analysis of data gathered about the MOTHER system.

# Mother2 and MOSS: Automated Test Generation from Real-Time Requirements

*Joe Maybee*
*Tektronix, Inc. / Portland State University*

## I. Introduction

Test fixtures can be valuable tools as part of a comprehensive QA strategy [1,2]. Fixtures can buy valuable time during the endgame phase of a project. However, fixtures are only the beginning of a strategy to build an effective QA process.

The Graphics Printing and Imaging Division's (GPID) print engine QA group is involved in the evaluation of real-time process control firmware for desktop color printers. Our QA evaluation process is geared toward the fast evaluation of printer firmware. Since we refer to the firmware as "software" in our group, I will be using the word "software" throughout this paper.

In order to improve GPID's QA process, we are engaged in constant evaluation of both strategies and tools. Having evaluated our previous successes and failures, we have installed incremental improvements in our process. In the case of our test fixture, we have redesigned and re-implemented MOTHER[1] [1] to eliminate the shortcomings in the original fixture.

The original MOTHER fixture allowed us to run large numbers of tests in a predictable amount of time. In addition, the original MOTHER tests were embedded in the requirements specification to allow easy maintenance of requirements and tests.

Our original system allowed our QA group to focus on maintenance of requirements, rather than exclusively focusing on the maintenance of tests. Since we were able to maintain requirements and tests efficiently and to run tests quickly, we turned to the problem of writing requirements and generating tests directly from these requirements. This paper describes the second generation test fixture (Mother2) and the requirements specification system we developed to generate tests directly from the requirements (MOSS)[2].

---

1. MOTHER stands for "Maybee's Own Test Harness for Evolving Requirements"

## II. A Key QA Contribution: Fast Evaluation of Software Releases

Fast evaluation of software releases is an important QA contribution in the final phase of a project. In the following sections, I will discuss the reasons for this and how our organization has supported fast evaluation in the past.

### The Key Project Phase: "Endgame"

The problems faced by QA engineers are almost universal. Specifically, the environment in which they work is chaotic and stressful. In particular, requirements are not always well considered and are volatile: change to the requirements can occur on an hourly basis in some environments. In addition, there is a great deal of pressure on the QA team to produce results quickly.

It is a fact of life that requirements are not always well considered. In many companies, market competition produces time pressure that is extreme. These time pressures frequently translate to pressures to reuse requirements that were incomplete, or to skip writing requirements altogether.

Today the belief that requirements can be "frozen" has fallen by the wayside. Markets have become far too volatile and unstable to coexist with long product development cycles. It has become necessary to be able to change product features in midstream easily and effectively.

The endgame phase of a software project brings with it higher levels of stress. Like the endgame in a chess match, it is the point in the game where you have either prepared yourself to be in the winning position or you have lost the game. In a chess game it is usually possible to recognize winning or losing endgame positions. In a software endgame, winning positions are defined by whether you are prepared to discharge your duties and responsibilities in an effective and efficient manner.

A losing position in a software endgame is usually recognized by who is getting most of management's attention. In all fairness to management, their close attention at the closing phase of the project is not entirely arbitrary. Typically, advertisements have been bought at great expense, usually at least three months prior to publication. Also, in many instances, release dates have been chosen to coincide with conferences and expositions. Floor space in the vendors section has been rented, and is usually quite expensive. Finally, advance sales contribute to the endgame pressure.

In light of these factors (advertisements, conference schedules and advance sales), there is little or no surprise that pressure escalates as the final ship date approaches.

---

2. MOSS stands for "Maybee's Own Specification System". I hope that future tools will be named after someone or something else.

## The QA Organization and the Project Endgame

A good QA organization will understand and support the schedule in the most efficient manner possible. In short, QA will keep itself off the critical path for the software. At the end of the project, the software is typically passed from design engineering to QA in periodic releases. After a release of the software has been given to QA, management attention turns to QA. When the attention of management is focused on QA, it is best if the QA process runs like a well-oiled machine.

QA can be most effective during the closing phase of the project by producing comprehensive test results quickly. Project managers are under a great deal of stress after a release goes out, and some of that stress can best be relieved by a quick enumeration of defects. In our organization, we strive to provide complete functional testing within 48 clock hours of a release.

By “complete functional testing” we mean that all test suites that test functionality will be run within 48 hours. After the initial test cycle, release testing needs to answer two basic questions for management:

1. Did design engineering fix the defects they claim they fixed?
2. In the course of fixing defects or adding features, did the design team break anything that worked previously?

These two questions are best answered by checking test results with previous runs. Consequently, QA needs to be prepared with complete test suites before the first release, in order to establish a baseline of results for future comparisons.

## Our Old Method: MOTHER

The original test fixture (MOTHER) demonstrated that fast and effective turnaround was best served by an automated test fixture. MOTHER also demonstrated that test fixtures are powerful tools for serving the project endgame strategy outlined above [2].

With the original MOTHER fixture, we embedded the tests in the requirements document so that when a requirement changed, the test was easy to locate and change as well.

It became apparent that there were problems with the MOTHER approach, however. While MOTHER supported the real-time aspects of testing reasonably well, the sequential nature of its test language did not support changes to timing relationships. Consequently, the burden for calculating time differentials was placed on the test writer. In addition, the *sequence* of operations in the time domain had to be determined by the test operator. If a response to a system stimulus was a series of operations, the test writer had to determine the order of these operations, and calculate the time differential between each of the operations.

Timing relationships between the operations became a problem to maintain, especially with the number of tests that were written. In one instance, a timing relationship change caused a significant delay while we updated the tests: we were not able to run complete

tests for 10 days. We learned a valuable lesson from this: we were not serving the test writers needs with the current MOTHER language. Luckily, this happened only once on the original project.

## III. Our New Approach

We decided to take what we had learned from the original MOTHER fixture and specify, design, implement and test a new fixture: Mother2.

Mother2 was designed to implement the concept of formal real-time requirements in a fashion that would be easy to maintain. In addition, plain English text requirements could be generated from the formal notation for review by engineers not familiar with the formal notation.

### What is a Real-Time Requirement?

We used Real-time finite state automata (RT/FSA) as the fundamental model for our product software. Real-time FSA is one of the most natural and straight forward requirement methods for real-time reactive systems.

We may consider an FSA requirement a series of facts about:

- Initial State for this requirement. That is, the state that the system is in at the beginning of this requirement.
- Stimulus applied to the system.
- Response of the system to the state.
- Final state of the system. That is, the state that the system is in after the response to the stimulus is complete.

Since each and every requirement should have a unique identifier, we could write an FSA requirement as a five-tuple:

```
Requirement (Unique_id, Start_state, Stimulus, Response,
    Final_state).
```

However, this is insufficient for real-time requirements.

With real-time systems, we consider time as an unconstrained input to the system. In real-time systems we measure this input via a timebase, usually a real-time clock or deadman timer. Time becomes a critical factor in real time systems and Jaffe, et al. suggest that every input and output in a requirement should be constrained by a time window.

Consider the following example requirement for a Phaser 300i color printer:

- Initial State: ready.
- Stimulus: Media Eject (output) sensor is blocked [0 to 1000 mSec].

- Response: Transition to final state.
- Final State: fault_jam state achieved within 2000 mSec.

In this instance, the system response is covered by its transition to the final state. That is, nothing out of the ordinary is required of the system other than to attain the required final state. In the notation of our requirement system we could write this requirement in a fashion very similar to the form previously mentioned:

```
req(ready_12,ready,output_sensor(blocked),nothing,
    fault_jam, t2sec).
```

The requirement above follows our original requirement format with an additional parameter at the end of the list. This additional parameter specifies how long the system has to make the transition to the final state.

We refer to the stimulus as the *input predicate* and the system response as the *output predicate.* In the above example the input predicate is a single input with a simple value: output_sensor takes on the value “blocked”. The output predicate is “nothing”, indicating that the requirement is satisfied if the final state is achieved and nothing else.

### Maintaining the Time Relations

It was apparent that we needed to make the Mother2 fixture intelligent enough to determine what the timing relationships should be. This feature would allow the test writer to specify in a single place what the expected time for a response should be, and to compose complex responses based on combinations of these times. If the time should change, it can be revised easily, and the new timing relationships would be automatically calculated by the Mother2 fixture.

### Checking the Requirements

Previously, the only way to check for errors in the test itself (i.e. semantic or syntactic errors) was to run the test. The original MOTHER fixture would then flag problems with the tests. This was considered to be time consuming effort. We designed our new requirements specification system (MOSS) to allow for a test “prescan”. Since complex responses are built up from elementary “building blocks” (inputs and outputs arranged in time), it is possible for us to check these complex responses. In order to “prescan” the test, we simply need to check that every requirement in the system is composed of valid components: valid states, stimuli and responses. Since system stimuli are composed of inputs, outputs and specified times, these in turn can be checked to ensure that their basic elements have been specified.

For maintenance purposes we decided to modularize the knowledge bases for the requirements. By allowing the test writer to build and maintain separate knowledge bases for the requirement, we distributed the knowledge base across units of manageable size. For instance, the inputs, outputs and their valid values could be maintained in one

knowledge base file, while the combinations of outputs that compose a system “response” could be put in another knowledge base file, etc.

This new requirements specification system had the capabilities of emitting Mother2 tests as well as English language requirements from the new requirements database. We prefixed each Mother2 test with the English language representation of the requirement which produced a nice “header” for the test. (See the example test later in this paper.)

In short, we eliminated the test writers and turned them into requirements writers. We built a primitive expert system that would analyze the requirements, report errors in the requirements and generate tests. Since the tests were generated automatically, we were able to use the effort of our test writing team entirely for support of our requirements specifications.

Our new requirements specification system (MOSS) was written in Prolog.

## The Power of Prolog

Prolog is a language that supports logic programming. Logic programming is a powerful tool for establishing relationships among facts in a knowledge base. We found Prolog to be a very natural tool for dealing with our requirements for several reasons:

1. Requirements are a collection of assertions about the desired behavior of the product software.
2. Requirements are composed of facts about the system (i.e. inputs, values of inputs, outputs, values of outputs, etc.).
3. Problems with the requirements can be detected using a set of rules. For instance, if a requirement has in its input predicate something that cannot be identified as an input, or a list composed of inputs, then we have a malformed requirement. Prolog allows us to make the assertion that a particular parameter is a fact (already listed in the Prolog fact base), and to detect exceptions to this assertion.
4. Tests can be generated using a set of rules on how to translate facts about requirements into the Mother2 test language.

## The Problems of Prolog

Prolog, however is not without it's drawbacks. In particular, we encountered three types of problems:

1. Database order dependencies.
2. Range problems.
3. Data Typing.

It takes some knowledge and experience to write effective Prolog programs. Since we were terribly inexperienced with Prolog, we were capable of coding some unusual but fascinating bugs into MOSS. Our greatest problem was that we were unable to locate any Prolog experts to answer our questions. However, after some time and research we were able to come up with several excellent sources dealing with logic programming guidelines and Prolog style. [4,5,6]

### *Database order dependencies*

Some of the major problems that we encountered were order dependencies in the fact base. Prolog has strict rules about the evaluation and search order of the Prolog database, and the order in which rules and facts are set down can have consequences. Knowledge of good Prolog programming style will prevent many of these problems.

### *Range problems*

Prolog cannot solve certain problems well. In particular, the nature of the language suggests that all assertions based on the rule base can be dealt with explicitly. With Prolog, this is not entirely true. Dealing with constraints, such as constrained numeric ranges, requires knowledge of Prolog's limitations. CLP(R) is a language that has been developed which deals with these problems in a much more natural fashion. Any QA team building a MOSS-like requirements specification system may wish to investigate CLP(R).

### *Typing*

Prolog is not a strongly typed language. If data is to carry type information, that type information must be built into the database. This explicit typing requires more information to be built into the fact base, and adds to the complexity of the Prolog database. For example, if output_sensor is a valid system input, a fact to that effect must be present in the database for purposes of validating the predicates:

```
system_input(output_sensor).
```

We have determined that a typed language, such as GOEDEL is a viable and desirable alternative to Prolog for solving this particular problem.

## Composing Requirement Building Blocks

We have already stated that we wished to have automatic evaluation of time relationships determined by the Mother2 fixture. Typically, these times are significant to many different requirements.

We use “time tags” to describe these significant time intervals with a symbolic name. The general form of the time tag is:

```
time(tag_name, T_zero, Delta_T, ‘time units’ ).
```

For example, the amount of time to wait for a power-up event may be described as “pwr_time”, and information about it could be contained in the Prolog fact base:

```
time(pwr_time, 200, 100, ‘ms’ ).
```

This “time tag” can now be used in our requirements and need not be explicitly stated. Wherever we use the “pwr_time’’ tag, our system will know that we are talking about an interval beginning at 200 mSec and continuing to 300 mSec (the 100 mSec is a delta-t from the beginning of the interval.)

We can now use these symbolic names to assemble “time lists”, which are lists of time constrained events.

### *Building a Time List*

If the time required for a power-up event should ever change, we can change the time tag description in the database, and the time will change throughout the requirements database.

In addition, when composing lists of events, these events need not be in time order. Consider the following “initial state list”:

```
initial_state_list(ready,
[
    [reset_system, ‘push’, push_time],
    [reset_system, ‘free’, free_time],
    [system_pwr, ‘on’, pwr_time],
    [front_panel, ‘”AReady-- “‘, t20min]
]
).
```

This “initial state list” contains a specification on how to walk the Phaser 300i printer into a state we have named “ready’’:

1. Set the Mother2 “reset_system’’ output to “push’’ at “push_time’’.
2. Set the Mother2 “reset_system” output to “free” at “free_time”.
3. Set the Mother2 “system_pwr” output to “on” at “pwr_time”.

4. Wait for the Mother2 input “front_panel’’ to attain the ASCII value “Ready--’’ during the interval “t20min’’.

The Mother2 system contains a scheduler that will arrange these items in time order. So, we need not be concerned with whether the events occur in the order specified: The Mother2 fixture will impose the proper time order just before run time.

Many parts of a requirement can be composed of these “time lists”. Therefore, our requirements are composed of:

- *“Initial” state lists:* a list of events (and their times) that must occur to “walk” the system into the “initial” state for testing.[1] The t-zero for our tests occurs immediately after the Mother2 fixture has verified that we have achieved our initial state.
- *Input predicate lists:* a list of events that describe a stimulus (and their times) that should be applied to the system after the “initial’’ state has been reached.
- *Output predicate lists:* a list of events that describe a system response (and their associated times) that should be expected from the system after the input predicate has been applied.
- *Terminal state lists:* a list of values for outputs that should be expected in order to verify that the system has achieved its terminal (final) state.[2]

### *How Tests are Generated from Time Lists*

As one might expect, the generated tests for the Mother2 fixture follow the natural progression of the requirement:

1. *Walk the system into it’s initial state, using the information in the initial state list.* If the system fails to achieve the initial state, a failure exists.[3] If the initial state cannot be achieved, the test is flagged as failed, and Mother2 proceeds to the next test.
2. *Apply the stimulus to the system as defined in the input predicate list.* The time orders of the stimuli and responses are determined by the Mother2 fixture.

---

1. “initial state” does not imply the initial state for the system, rather it is intended to denote the point of departure. Therefore, our initial state could be a ready state, error state or any other state in our requirements specification.

2. As with the “initial state”, “terminal state” refers to the state the system should arrive at after the response is complete. The “terminal’’ state could be the ready state, the power-up state, or any other valid system state.

3. A printer may fail to achieve a state because it has exhausted its consumables: the paper tray may be empty, or the printer may be out of ink. This does not imply a failure of the requirement, it means that the printer needs attention. In this instance Mother2 will “ask” the operator to correct the problem and then try again.

3. *Watch the system to see that the response from the system occurs as defined in the output predicate list.* If any of these responses fail to occur within the times specified in the output predicate, the test is flagged as failed, and Mother2 proceeds to the next test.
4. *Check to see that the terminal state is achieved as defined in the terminal state list.* If the terminal state cannot be achieved in the time specified, the test is flagged as failed, and Mother2 proceeds to the next test.

This follows the natural sequence of the requirement specifications: state, stimulus, response, final state.

We have reduced the elements of the requirements into collections of elementary inputs and outputs arranged in time. These collections of inputs and outputs are grouped into test "phases": initial state, stimulus, response and terminal state.

Consequently, there are only two basic operations that the Mother2 fixture needs to perform in order to test our real-time system:

1. Set an input to the system under test to a specific value within the time interval specified.
2. Check an output from the system under test to see that it achieves a specific value within the time interval specified.

These two fundamental operations translate into another two basic Mother2 operations:

1. "tset'': set a value to an input in a specified time window.
2. "twaitfor'': wait for an output to take on a specified value in a time window.

In addition to these two basic operations, information had to be grouped into what we called "timegroups''. For instance:

- *stategroups:* state groups were groups of Mother2 commands that would be re-tried (at the operators discretion) if they failed. stategroups were used to achieve the initial state. stategroups are delimited with "statestart" and "stateend" keywords.
- *timegroups:* these are time critical groupings of tsets and twaitfors that may not be retried if they failed. Input and output predicates fall in this domain, as do terminal state checks.

### An Example Requirement and Its Generated Test

Here is an example of a requirement, it's associated Prolog database, and the generated test:

*Requirement*:

```
req(ready_12,ready,output_sensor(blocked),nothing,
    fault_jam, t2sec).
```

*Ground base definitions:*

```
system_input(output_sensor).
system_input(reset_system).
system_input(system_pwr).

system_output(front_panel).
system_output(fault).
system_output(centronic).

output_sensor(X) :- member(X , [blocked, notblocked,
    release]).

time(push_time, 0, 100, 'ms' ).
time(free_time, 100, 100, 'ms' ).
time(pwr_time, 200, 100, 'ms' ).
time(t20min, 10000, 1200000, 'ms').
time(t10_10,10000,10000, 'ms').
time(t02sec, 0, 2000, 'ms').
```

*State list definitions:*

```
initial_state_list(ready, [
    [reset_system, 'push', push_time],
    [reset_system, 'free', free_time],
    [system_pwr, 'on', pwr_time],
    [front_panel, '"AReady-- "', t20min]
 ] ).

terminal_state_list(fault_jam,
 [
    [fault, 'low', t02sec],
    [centronic, '"Paper_Jam SET"', t10_10]

 ] ).
```

*Generated Test:*

```
//
// Requirement: ready_12
//
// State: ready
//
// Stimulus: Input output_sensor takes on value blocked
//  at 0 ms
//
// Response: nothing
//
// Requirement to be met within the time period
// starting at 0 ms and ending at 2000 ms.
//
// Next state: fault_jam
//
//

teststart ready_12

statestart
tset 0 100 SUTReset push
tset 100 100 SUTReset free
tset 200 100 SUTPwr on
twaitfor 10000 1200000 LCDLine2 “AReady-- “
stateend

timestart
tset 0 1000 MediaEject blocked
timeend

//
// Check final state
//

timestart
twaitfor 0 2000 Fault low
twaitfor 10000 10000 Centronics “Paper_Jam SET”
timeend

//
// ************* End of Test *************
//

testend
// testend: ready_12
```

## IV. Evaluation of Effectiveness

In the process of evaluating our next generation system, we ran the original MOTHER test fixture in parallel with the Mother2 fixture. This comparison was intended to evaluate and compare the effectiveness of Mother2. Mother2 was allocated a subset (approximately 80%) of the requirements to test in parallel with the MOTHER test fixture.

Turnaround time for these tests was 30 clock-hours on a single MOTHER fixture, and 10 hours for two Mother2 fixtures, which would translate to approximately 20 hours for a single Mother2 fixture. Therefore, we may consider Mother2 more efficient than the original MOTHER.

All defects found by the MOTHER fixture were also found by the Mother2 fixture (for the 80% of the requirements that were in the common requirements subset). This was a great confidence booster for us.

On one occasion, a wide-ranging timing change was required (a timing change which affected a large number of requirements), and this timing change was made in the MOSS requirements specification in less than an hour. The original MOTHER tests required about week of modification before the tests were properly revised.

However, not everything worked well. We discovered that in building our input predicates, it would have been nice to use other predicates as “building blocks.” In short, we would have liked to write predicates by telling MOSS to use other predicates as departure points, and to incorporate some additions.

The Prolog language is difficult for requirements writers. Prolog is a somewhat arcane language, and we need to liberate our requirement writers from having to know Prolog.

We are in the process of incorporating the following changes to our Mother2/MOSS facilities:

1. We are writing a front-end to MOSS which will allow us to generate Prolog databases from a much looser requirement specification form.
2. We are in the process of installing facilities in MOSS which will allow us to build predicates as aggregates of other predicates.
3. We are in the process of trying to translate criteria laid down by Jaffe and Leveson, et al. into Prolog clauses. This will automate the analysis of our requirement specifications.

I hope to report back about the problems and successes of these efforts at a future conference.

## V. Acknowledgments

I would like to take a moment to acknowledge the people who were essential in making this paper possible.

First and foremost, my manager, **Raul Krivoy**. Raul's abiding faith in our abilities and his undying commitment to quality have been the key to our successes. Thanks also to my friend and colleague, **Gene Welborn**, who is the coauthor, codesigner and coimplementor of the Mother2 portion of our system. **Harry Ford** translated lots and lots of requirements into the new formats,[1] built the knowledge bases, and managed to keep both his sanity and humor. **Richard Waugh** managed to translate vague descriptions of what we were trying to accomplish into concrete implementations of hardware interfaces. **Mike Rogers**' improvements to the MOTHER fixture gave us an ever-improving standard of quality to try and beat with the Mother2 fixture. Last, but not least, my wife **Jan Maybee**, who proofread the initial and final drafts and entertained our two-month old daughter, **Annie**, while I wrote this paper.

## VI. References

[1] Maybee, Joe, *MOTHER: A Test Harness for a Project with Volatile Requirements.* Proceedings of the Pacific Northwest Software Quality Conference, Portland Oregon, 1991.

[2] Maybee, Joe, *True Stories: A Year in the Trenches with MOTHER*, Proceedings of the Pacific Northwest Software Quality Conference, Portland, Oregon, 1993.

[3] Jaffe, Matthew S, et al. *Software Requirements Analysis for Real-Time Process-Control Systems,* IEEE Transactions on Software Engineering, Vol. 17, No. 3, pp 241-258, March 1991.

[4] Ross, Peter, *Advanced Prolog*, Addison-Wesley, New York, 1989.

[5] Coelho, Helder and Cott, Jose C., *Prolog by Example*, Springer-Verlag, Berlin, 1988.

[6] Knuutila, Timo, *Efficient Prolog Programming*, Software-Practice and Experience, Vol. 22(3), pp 209-221, March 1992.

1. Actually, Harry translated ALL of the requirements used by MOSS and Mother2.