Performance and Dependability Modeling with Möbius

Shravan Gaonkar[1], Ken Keefe[1], Ruth Lamprecht[2], Eric Rozier[1],
Peter Kemper[2], William H. Sanders[1]
[1]University of Illinois, Coordinated Science Laboratory, Urbana, IL 61801, USA
[2]College of William and Mary, Department of Computer Science, Williamsburg, VA 23187, USA
{gaonkar, kjkeefe, ewdr, whs}@crhc.uiuc.edu, {rlampy, kemper}@cs.wm.edu

ABSTRACT

Möbius is a multi-paradigm multi-solution framework to describe and analyze stochastic models of discrete-event dynamic systems. Möbius is widely used in academia and industry for the performance and dependability assessment of technical systems. It comes with a design of experiments as well as automated support for distributing a series of simulation experiments over a network to support the exploration of design spaces for real-world applications. In addition to that, the Möbius simulator interfaces with Traviando, a separate trace analyzer and visualizer that helps to investigate the details of a complex model for validation, verification, and debugging purposes. In this paper, we outline the development of a multi-formalism model of a Lustre-like file system, the analysis of its detailed simulated behavior, and the results obtained from a simulation study.

1. INTRODUCTION

In model-based system design, Möbius [1] focuses mainly on discrete-event dynamic system models (in contrast to continuous simulation models based on differential equations) and within that area, rather on event-driven stochastic automata than a process interaction approach. One advantage of Möbius is the ability to integrate the evaluation of performance and dependability of a system, emphasizing any dependencies that exist between these two aspects of system behavior. This is possible due to the ability to specify dynamic behavior of models in Möbius using a collection of state variables, actions, and rewards that can quantify the measures of interest.

In Möbius, a *model* is a collection of state variables, actions, and reward variables expressed in some formalism. Briefly, *state variables* hold the state information of the model. *Actions* change the state of the model over time. *Reward variables* are quantitative measures of interest defined by the Möbius user to evaluate their models. A *formalism*, e.g. stochastic activity networks, is a language used for expressing a model within the Möbius framework.

The Möbius tool is built on the observations that no formalism is best for building and solving all models, that no single solution method is appropriate for solving all models, and that new formalisms and solution techniques are often hindered by the need to build a complete tool to handle them. Möbius addresses these issues by providing a broad framework in which new modeling formalisms and model solution methods can be easily integrated. Möbius currently provides a variety of numerical and analytical techniques for the analysis of specific Markovian models as well as discrete-event simulation as a technique that applies to a very general class of models.

Möbius has been licensed by over 100 academic institutions and 30 industrial partners and has been successfully applied to evaluate systems with respect to multiple system properties including reliability, availability, security, and performance, and in a multitude of areas including Information Technology Systems and Networks, Wired and Wireless Telecommunication Software and Hardware Systems, Aerospace and Aeronautical Systems, Commercial and Government Secure Information Systems and Networks, and Biological Systems.

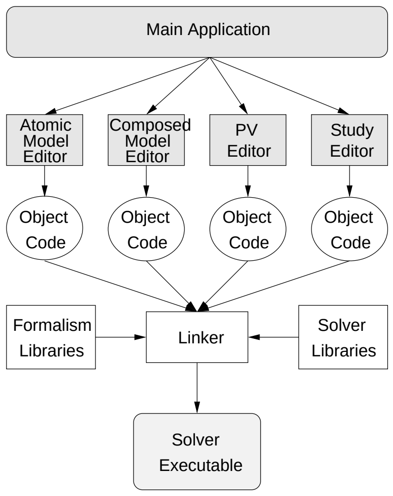


Figure 1: Möbius architecture from users perspective

Fig. 1 shows the Möbius tool architecture with its two distinct logical layers: model specification and model execution. All model specification is done through Java graphical user interfaces, and all model execution is done exclusively in C++. Each model editor produces C++ code that is compiled and linked with library classes to obtain an executable, model-specific solver, which gives great flexibility to support different domains and environments as well as good performance. In this paper, we will walk through the process of using Möbius as a modeling tool to effectively design, develop, and refine models of interest. Section 1.1

[1]Möbius : `www.mobius.uiuc.edu` Traviando: `http://www.cs.wm.edu/~kemper/traviando.html`

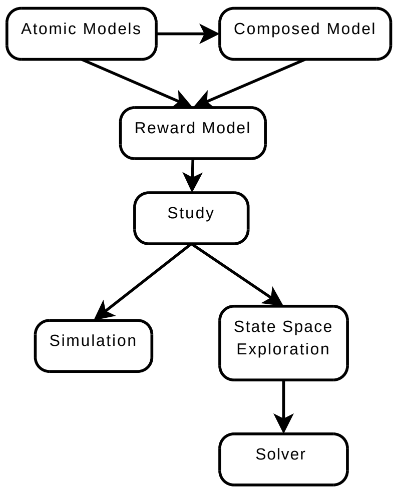


Figure 2: Möbius workflow dependencies.

reviews additional reference materials regarding modeling tools that readers might be interested in. In Section 2 we outline the generic process of modeling with Möbius. Section 3 provides a interaction between Möbius and Traviando with a concrete example of a model to illustrate the efficacy of multi-formalism tools and visualization tools.

1.1 Further Readings

A detailed description of the Möbius modeling tool is given in [2, 9]. The details of the Möbius framework and its implementation can be found in [3]. Several papers have been written using the Möbius tool which can be found at `http://www.mobius.uiuc.edu/community.html`. Möbius was developed as a natural extension of UltraSAN for its Stochastic Activity Networks formalism [10]. Several other formalisms have been integrated into Möbius: Modest [1], PEPAfault-trees [6], and buckets and balls [8]. Möbius supports several state representations and solution techniques. [4] provides further details on how to integrate state-level representation and solution techniques into Möbius. [11] gives in-depth understanding of how some of the numerical solvers work in Möbius. For information on Traviando, we refer to [7] and `http://www.cs.wm.edu/~kemper/traviando.html`

2. WORKFLOW IN MÖBIUS

Möbius clearly separates different aspects of a model into a set of components that depend on each other. Fig. 2 shows the dependency graph of components that a user develops while working with Möbius. Each individual node will be addressed in the following sections.

In the Möbius GUI, the project manager gives access to menus that allow the user to perform various tasks at the project level, such as creating a new project, opening an existing project, archiving/unarchiving a project, etc. From the Project menu, the user can easily create a new project by simply specifying a name for it.

The project window displays the skeleton of a Möbius model in the form of a tree. The six child nodes of the root (Atomic, Composed, Reward, Study, Transformer, and Solver) hold all the corresponding elements of the model and corresponds to the workflow model.

2.1 Atomic Models

The first branch of the tree in the project window holds all of the atomic models that have been defined in the project. At least one atomic model is required before the user can move on to lower branches in the tree.

There are a variety of atomic model formalisms, including stochastic activity networks (SANs), fault trees, and stochastic process algebra (PEPA). For further details on these formalisms or how it is possible to write a new one and use it with Möbius, refer to [9].

A major feature of Möbius atomic models is that they can be parameterized with the use of global variables. The global variables are later used to define a study, which can vary the values of global variables and generate a series of experiments to evaluate the model for different parameter values.

2.2 Composed Models

While it is not necessary to create composed models, they provide tremendous flexibility and power for modeling in Möbius. Composed models make it possible for the user to model systems of great complexity. Often, a modeler will create several atomic models that represent sub-systems of the total system. The Möbius composition formalism provides the flexibility and simplicity for the modeler to mix and match the atomic and other composed models together to build larger and more sophisticated models.

A Rep-Join composition formalism enables the modeler to either replicate or join atomic models or other composed models. Replication creates multiple instances of a single atomic or composed model and provides a way to share certain state variables across all of the replications. Joining simply joins two or more atomic or composed models. As with replication, it is possible to define certain state variables to be shared across the join.

One of the great strengths of Möbius composed models is that replicating and joining are agnostic of the underlying formalisms of the atomic models. So, if one object in a system is easier to express with a fault tree, it can be joined with another object that is expressed with a SAN as described by the example model in section 3.

2.3 Reward Models

Once the user has defined at least one atomic or composed model, a reward model can be created for it. A reward model is used to define measures of interest (performance variables) that the user wants to obtain from the system model. As the model is simulated or numerically solved, the reward model defines what data from the system needs to be collected.

A reward model can define several performance variables on an atomic or composed model. For each performance variable, the user inputs a segment of C++ code that defines the way the result is collected from the system. The user has the option of defining a rate reward, which defines its reward based on time in each state of the system, or an impulse reward, which defines its reward based on the firing of specific actions in the system.

The results can be collected in many ways in relation to time: at a specific instant of time, over an interval of time, over a time-averaged interval of time, or after the system reaches steady-state. Also, it is possible to estimate the mean, variance, likelihood of results in a range, and distribution of results for each performance variable.

2.4 Studies

A study allows a user to specify a series of experiments to be performed on a parameterized model in an automated manner. It is a productivity enhancer for running large sets of production runs of simulation models on a network of machines. A design of experiments feature is offered to reduce the combinatorial explosion that takes place if multidimensional design spaces need to be explored.

Note from Fig. 2 that a study is required in order to generate the state space of a Markovian model or to run a simulation. If the user does not wish to vary any part of the system, a trivial empty study can be used to move on to the next step.

2.5 State Space Generators and Solvers

A numerical analysis of a Markovian model is performed in two steps, a state space exploration and the numerical calculation of a transient or steady state distribution which is in turn used to evaluate rewards of interest. For the task of the state space exploration, Möbius provides two variants, one results in a sparse matrix representation of the associated Markov chain, the other achieves a symbolic representation based on matrix diagrams, which is very space efficient for compositional models that yield extremely large equation systems [4].

For a subsequent numerical analysis, Möbius provides a large array of solvers, each with their own strengths and weaknesses. Möbius has a direct steady-state solver, an iterative steady-state solver, a Takahashi steady-state solver, a transient solver, an adaptive transient solver, an accumulated reward solver, a deterministic iterative steady-state solver, and an advanced deterministic iterative steady-state solver [9].

2.6 Simulator

For a given model, reward, and study, Möbius can perform a stochastic discrete event simulation to obtain results. The simulator window allows the user to define the minimum and maximum number of batches to constrain the extent of a simulation. Within these bounds, the simulation proceeds till its result converge to the confidence interval defined in the reward model. The simulator window also has several options for the types of trace files to generate and what level of detailed information to output in a trace. These options are of relevance when Möbius is used in conjunction with Traviando, as described in section 2.7.

The Möbius simulator allows experiments to be run in a parallel fashion. If the user's machine is a multi-processor machine, the Möbius simulator can make good use of each processor. Within a network of machines, Möbius can distribute the workload across those machines by replicating the experiments to run independently. The results from those machines are combined by the Möbius Java client. This has the potential to dramatically reduce the running time of an extensive series of experiments.

2.7 Verification, Validation, and Debugging

A powerful framework like Möbius makes it easy to obtain quantitative results from large and complex models. However, for certain steps in a simulation study there is a need to check the dynamic behavior of a model at various levels of detail; for instance, to verify if the dynamic behavior of a model matches with a conceptual model (verification) or a real system (validation), or simply to track down root causes of observed phenomena if a modeler believes errors are present in the model. For this purpose, Möbius is able to export detailed trace data of a simulation run into a trace file for subsequent analysis in Traviando, a trace visualizer and analyzer. Traviando's functionality include statistics on state variables and actions, analysis of repetitive and progressive fragments in a trace, reduction operations to illustrate how a selected state is reached, LTL model checking to identify locations where particular properties are violated and many more. Möbius traces are visualized with message-sequence-chart-like graphics that resemble the compositional structure of Möbius models.

2.8 Refining The Model and Reporting Results

In practice, building a model is never a single-pass process. From any point in the workflow, a user can go back to retrace previous steps to alter the existing models or add new features to the model. However, it is important to resave items at the same or lower levels in the dependency graph so that any changes that have taken place will propagate down to the solvers and/or simulator.

Once a model is in a stable state, a user often wants to present the model definition along with the resulting data. Möbius provides a documentation feature that allows users to generate PNG, SVG, and PDF diagrams of the various models in the system. The document feature will also generate an HTML file that details the inner workings of each of the elements in a model (rates, case distributions, activation predicates, etc.).

Möbius is also able to interface with a PostgreSQL database for the purpose of storing results from simulation and solver executions. Of course, the data is available for use with any tool that can interact with a PostgreSQL database. In addition to that, the Möbius team is currently developing a data visualization tool for researchers to manage and use their database repository of Möbius results.

3. MÖBIUS WITH AN EXAMPLE

In order to demonstrate the use of the Möbius tool in a realistic example with multiple formalisms, we present a simplified model of a Lustre-like file system [5]. Our model consists of a single metadata target (MDT) and five object storage targets (OSTs), with failure models for each component. In a Lustre-like file system, the MDT stores information on directory compositions, and attributes for file data, such as permissions, as well as file layout. OSTs are then responsible for storage and retrieval of the file data itself. To illustrate the concept of modeling with different formalisms, the model is composed using Rep-Join from SAN, PEPA, and Fault Tree models. In sections 3.1, 3.2, 3.3, and 3.4 we will describe models for the subsystems of our Lustre-like file system utilizing these different formalisms, combining these atomic models in section 3.5 by sharing state-variables via equivalence sharing.

3.1 MDT Model

The first part of our model is a simplified MDT (Fig. 3). The primary purpose of this part is to model the processing of Input/Output (I/O) requests as they enter the file system. The action `IORequest` produces tokens in the `RequestQueue`. The lack of incoming arcs represents that it draws its re-

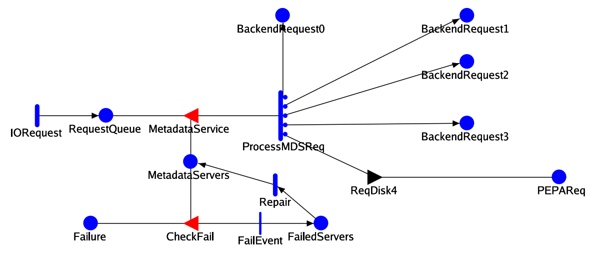


Figure 3: Metadata Server Model using SAN

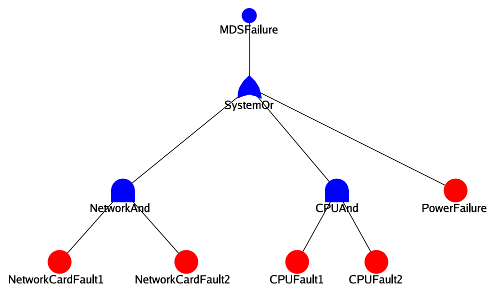


Figure 4: Metadata Failure Model using Fault-trees

quests from a theoretically infinite population, and thus the rate of incoming requests is not changed by the state of the model itself. Once a token is added to the place `RequestQueue`, it is serviced provided the server is not in a failed state (a condition checked for by the input gate `MetadataService`). Once serviced, the cases of the `Process-MDSReq` activity move the request to the appropriate OST queue, via either the shared `PEPAReq` place for the PEPA OST or one of the shared `BackendRequest`i places for the SAN OSTs. The rates of the activities in this model are provided by global variables, allowing the possibility of modifying the rates during analysis.

A portion of the MDT failure is also simulated in this model. The fault tree model described in section 3.2 shares the `Failure` place, allowing this SAN model to trigger failures with the instantaneous activity `FailEvent`, provided a server is in the working state (checked for by the input gate `CheckFail`). When this failure happens, both the failure token and a metadata server token are removed, and a token is added to `FailedServers`. Repair is modeled by moving tokens from `FailedServers` to `MetadataServers`.

3.2 MDT Failure Model

A fault tree is used to model the failure of metadata servers at a greater detail than a simple failure rate. The fault tree used here (Fig. 4) models a single MDS that can have failures in two network cards, two CPUs, or a single power supply. Network cards and CPUs are redundant so they can suffer one failure without taking down the system, but if both network cards, both CPUs, or the power supply fail, the fault tree will record a failure event in the head node `MDSFailure`, which is shared with the `Failure` place of the model described in section 3.1.

```
pr := 0.003;
rr := 0.0003;
fr := 0.0000004;
ProcessRequest[a] = [a > 0] =>
                    (outa, pr).ProcessRequest[a-1];

Failure = (outa, T).Failure +
          (fail, fr).(recover, rr).Failure;

System = ProcessRequest[0] <outa> Failure;
```

Figure 5: OST Model using PEPA

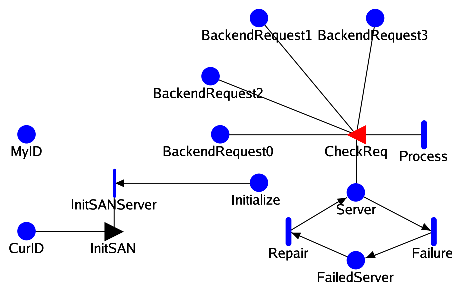


Figure 6: OST Model using SAN

3.3 OST PEPA Model

The process algebra PEPA is used to model one of the OSTs (Fig. 5). Two processes, `ProcessRequest` and `Failure`, are synchronized on the action `outa` to model the availability/unavailability of the OST due to failure. Because the `Failure` process can evolve into an unnamed process on the `fail` action and then recover to `Failure`, it is not always available to synchronize with `ProcessRequest` on `outa`, representing a period where `ProcessRequest` cannot remove tokens from the variable `a`, or a failure of the server. Otherwise `ProcessRequest` can consume tokens from `a`, representing the server fulfilling I/O requests. The variable `a` receives new jobs via state sharing with the MDT model's place `PEPAReq`.

3.4 OST SAN Model

The SAN model of an OST (Fig. 6) consists of two main portions: the first is an initialization and the second is how requests are processed. The initialization section uses the instantaneous activity `InitSANServer`, along with the place `Initialize` and shared place `CurID`, to set the place `MyID` to a unique index for each instance of the model. Each of the SANs in turn fire `InitSANServer`, automatically incrementing `CurID` while assigning the current value to `MyID`, which is used to access the correct `BackendRequest`i.

The other section of the model is functionally equivalent to the model in section 3.3. A server is in either a failed state or not-failed state, and transitions to the failed state eventually repairing. When not failed, it can consume I/O requests using the activity `Process`.

3.5 Composed Model

The entire model is composed of the submodels described in the preceding sections using the Rep-Join formalism (Fig. 7). First a set of three MDT Failure models are replicated, shar-

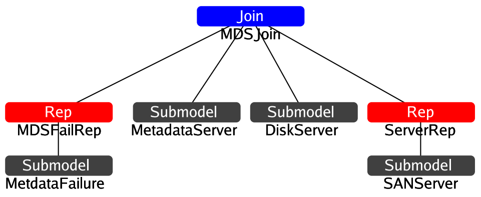
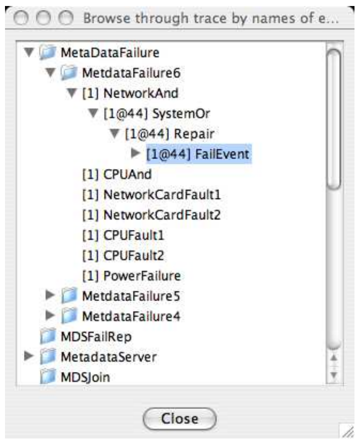


Figure 7: Composed Model



```
▼ MetaDataFailure
  ▼ MetdataFailure6
    ▼ [1] NetworkAnd
      ▼ [1@44] SystemOr
        ▼ [1@44] Repair
          ▶ [1@44] FailEvent
    [1] CPUAnd
    [1] NetworkCardFault1
    [1] NetworkCardFault2
    [1] CPUFault1
    [1] CPUFault2
    [1] PowerFailure
  ▶ MetdataFailure5
  ▶ MetdataFailure4
MDSFailRep
▶ MetadataServer
MDSJoin
```

Figure 9: Event browers

ing their head node, which is then joined with the MDT model on the `Failure` place. Second,the OST PEPA model *DiskSever* is joined to the MDT model by sharing the `a` variable with the place `PEPAReq`. Finally, four OST SAN models are replicated, sharing the places `BackendRequest`i and `CurID`, with the places `BackendRequest`i being further shared with the MDT model.

3.6 Simulating the MDT Model

For space limitations, we do not provide details on how to specify measures of interest like mean time between failures. Möbius provides support for an ample variety of such measures based on rate and impulse rewards, which can be measured at any given point of time, in steady-state, accumulated over a period of time, or time-averaged over a period of time. For any global variables declared in atomic models, we can specify exact values or a set of values to be exercised with a series of experiments. The evaluation of a single experiment can be performed with discrete-event simulation and, for certain types of models, also with a numerical analysis of an associated continuous time Markov chain. However, simulation is the evaluation method most broadly applied in practice.

In this section, we focus on discrete-event simulation, particularly on support provided to investigate the details of a simulation run, which can be deemed necessary if the behavior of a complex model results in unexpected performance or dependability values, and a modeler wants to examine the detailed steps performed in a simulation run. The Möbius simulator has options to output a simulation run as a se-

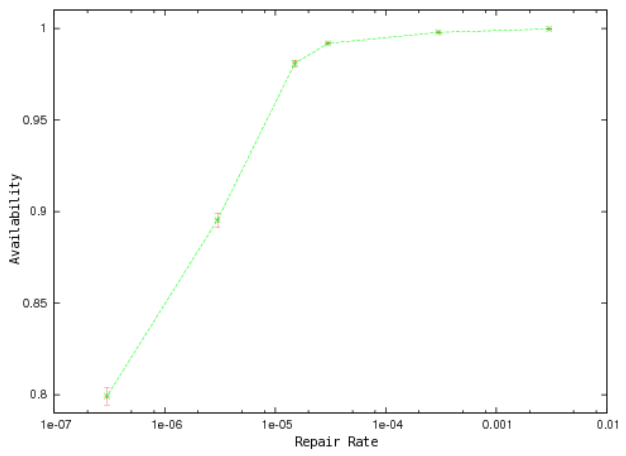
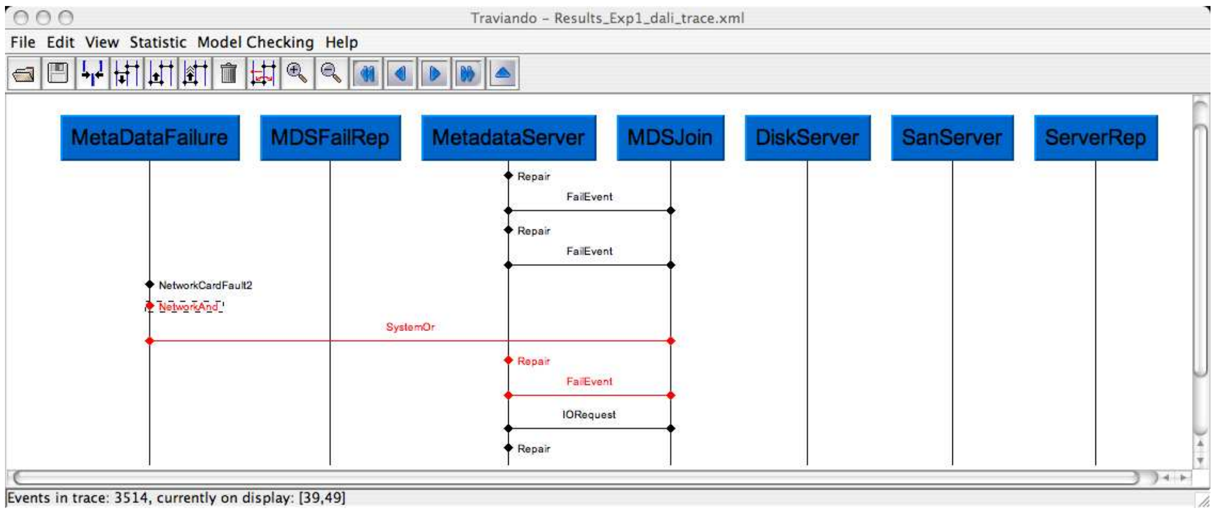


Figure 10: Mean availability of one or more Metadata Servers for varying repair rates, with 0.95 confidence intervals.

quence of states and events in an XML-formatted trace file that can be analyzed and visualized with Traviando. Traviando is a stand alone tool to visualize simulation trace and provide statistical as well as model-checking capabilities to support verification and validation of simulation models.

Fig. 8 shows a fragment of a trace that results from the simulation of the MDT model as a variant of a message sequence chart (MSC) or sequence diagram. The compositional structure is preserved by the representation – each process in the MSC corresponds to a node in the composed model shown in Fig. 7. Note that the simulator sees an n-times replicated atomic model like *MetaDataFailure* as n individual submodels, which implies n corresponding processes in the MSC. Traviando allows a user to group processes together, which can result in a tree type hierarchical organization of processes. This has been done in Fig. 8 to achieve a visualization of manageable size. Scalability of visualizations is a major concern in Traviando, since traces often result in large sets of variables (which constitute state information visualized in separate windows), large sets of actions (which are made accessible in a separate browser window), large sets of processes (which can be grouped to control the level of detail visualized in an MSC), and most dominating, a large set of events (which implies that only a trace fragment of configurable length is shown at a time). The length of traces implies that a manual investigation is tedious and tool support as given by Traviando is helpful. Traviando provides statistical data on variables and actions. Simulation models tend to reproduce sequences of events with certain regularities, most notably that certain states are reached in a repeated manner such that for debugging purposes it is interesting to obtain a reduced trace that results from the removal of cycles. There is a measure of progress that quantifies the length of a reduced trace. Plotting the progress meausure for each prefix of a sequence of events in a given trace shows distinctive patterns that are helpful to identify certain errors in simulation models, e.g., a constraint violation for a constant customer population in a queueing network model in [7]. Traviando also provides a feature to perform LTL modelchecking on a given trace and



Events in trace: 3514, currently on display: [39,49]

Figure 8: MSC

highlight those states in a trace that fulfill given properties.

A recently added feature in Traviando is a browser that helps a user select sequences of events of interest. Fig. 9 shows an interactive window of actions seen in the trace with corresponding cardinality and the possibility to expand the node in the window to see choices for successor events seen at various locations in the trace. In that figure, we selected an event *NetworkAnd* which occurs only once and is followed by events *SystemOr* and *Repair*. If the cardinality of locations for that sequence is 1, additional information on the particular location in the trace is given. A click on any event in the window will show a corresponding trace fragment in the MSC window, as it is the case for this example in Fig. 8. With all this information, errors can be located with Traviando and corrected in Möbius.

For a correct model, production runs to evaluate a large design space can be conducted with Möbius in a distributed manner. Results are collected in a database for further use. The simulation of the MDT model yields numbers to evaluate the dependability of the modeled file system.

Fig. 10 illustrates the use of a manual range study with our model. The range study is used to vary the repair rate of components in the system to illustrate the effect that varying the repair rate over five orders of magnitude has on the availability of one or more metadata servers. Steady-state simulation was used to estimate mean availability of one or more metadata servers, along with 0.95 confidence intervals. In this example one might imagine a system design team using Möbius to make a decision on the number of repairmen to keep onsite, based on the marginal benefit a reduction in repair time would yield.

4. CONCLUSION

The multi-paradigm multi-solution framework Möbius provides a comprehensive framework for a model-based dependability and performance evaluation of systems. Traviando is stand alone trace analyzer that provides complementary functionality to the verification and debugging of simulation models. Future work is dedicated to add more support for distributed simulation in Möbius and for runtime verification of distributed Möbius simulations by Traviando.

5. REFERENCES

[1] H. Bohnenkamp et al. On integrating the Möbius and Modest modeling tools. In *Proc. Dependable Systems and Networks, 2003*, pages 671–671, IEEE, 2003.

[2] G. Clark et al. The Möbius modeling tool. In *Proc. 9th Int. Workshop Petri Nets and Performance Models*, pages 241–250, IEEE, 2001.

[3] D. D. Deavours et al. The Möbius framework and its implementation. *IEEE TSE*, 28(10):956–969, Oct 2002.

[4] S. Derisavi et al. The Möbius state-level abstract functional interface. *Perform. Eval.*, 54(2):105–128, 2003.

[5] S. Gaonkar et al. Scaling file systems to support petascale clusters: A dependability analysis to support informed design choices. In *Proc. Dependable Systems and Networks*, IEEE, 2008.

[6] R. Gulati and J. B. Dugan. A modular approach for analyzing static and dynamic fault trees. In *Proc. Reliability and Maintainability Symposium. 1997*, pages 57–63, 1997.

[7] P. Kemper and C. Tepper. Automated trace analysis of discrete event systems models. *IEEE TSE*, (in print) 2008.

[8] R. L. Klevans and W. J. Stewart. *XMARCA: User's Manual*. Department of Computer Science, North Carolina State University, Raleigh, N.C. 27695-8206, USA, October 1992.

[9] Möbius Team. *The Möbius Manual*. University of Illinois, Urbana Champaign, Urbana, IL – 61801, 2007.

[10] W. H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. In *Proc. 8th Int. Workshop Petri Nets and Performance Models*, pages 2–9, IEEE, 1999.

[11] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, N.J., 1994.