Pattern Detection and Extraction from Systems and Security Simulation Models

Towards a Model-Checking Framework for Security Analytics

Brian Monahan and Nick Papanikolaou
Cloud and Security Lab, HP Labs
{brian.monahan,nick.papanikolaou}@hp.com

April 19, 2012

Abstract

In this paper we describe a method, and implemented prototype, for extracting high–level process models for systems modelled using a simulation framework (for illustration we use the Gnosis language and toolset). Our technique builds a finite state automaton that characterises one or more simulation runs of a simulation model by including in its states selected parts of the latter's execution traces. The intention is that the generated automaton reveals the high–level structure of the original model, without making reference to (or requiring knowledge of) the source code of that model. We discuss applications for this technique and identify several directions for further work.

1 Introduction

One of the greatest challenges in modern computing is that of understanding and reasoning about complex system behaviour. Simulation is one of the most widely used approaches for dealing with this challenge, and numerous different simulation techniques and tools have been developed for a wide range of domains. The effectiveness of any given simulation depends on the accuracy and amount of detail contained in the model used to produce that simulation; therefore, a great deal of skill and experience is required to develop simulations which actually provide meaningful information and insight into the possible behaviours of a system. Our focus here is not on the development of such models, but rather on the information that can be gleaned about system behaviour from *the results of several simulations*, even when there is very little (or even no) knowledge of the original model used to produce them.

We argue that one can extract meaningful finite-state models from the results of simulations, which can reveal patterns and structure that is not necessarily obvious in the original model used for these simulations. This claim is of fundamental importance, as it means that simulations may reveal more information about a system than one might think; furthermore, the techniques we present here can be generalised so as to apply to any other experimental setup in which system behaviour is contained in ordered logs or timelines, and in which it is desirable to detect common patterns and underlying structure.

The types of simulations for which the techniques we present in this report find greatest utility are those where there is inherent non-deterministic behaviour, and also explicit specification of the probabilities of particular actions/events occurring. It is these types of simulations which give several different, and hence interesting, results (specifically, they give rise to different traces; this term is defined below). In order to account for the probabilities of different action patterns in simulations, our technique for extracting such patterns assigns explicit probabilities to actions and events.

In order to fully understand every aspect of a given system model, one could program a tool to systematically and exhaustively generate *all* possible behaviours described by the model, and then perform checks of correctness and other properties on each and every such behaviour; this type of verification is known in the literature as *explicit-state model checking*. We envisage incorporating the techniques we describe in this report in such a tool in the future, but for our present purposes we focus on extracting patterns and structure from a finite (but not necessarily exhaustive) set of behaviours.

1.1 Terms Used In This Report

The observation of multiple simulation runs will yield different (pseudorandom, to be precise) results when the model used contains explicit (or implied) nondeterminism or probabilities for particular actions. In this report we will use the term *run* (or *simulation run*) to refer to a single execution of a simulation. The output of a run is a *trace*, namely, an ordered sequence of states. A *state* in this context is defined as a snapshot of the values of all the variables in the system model at a specified time instant during a simulation, along with any printed messages produced at that time.

1.2 Tool Support for Simulations In This Report

To demonstrate our extraction algorithm we will make use of the *Gnosis* toolset[1] [6, 7], a modelling and simulation framework based on Demos2k. While the details of the Gnosis syntax are not of great importance for this paper, it is worth mentioning some tool specifics as they will be useful later.

[1]For more information, see `http://www.hpl.hp.com/research/systems_security/gnosis.html`.

When a system model named `model.gn` is input to the Gnosis tool, Gnosis will perform a single simulation run and print out the trace on the standard output and in the file `model.tr`. Other files produced by Gnosis during the simulation will include `model.csv`, `model.seed`, and `model.log`. The file `model.csv` contains snapshots of the overall state at specified instants during a simulation, namely whenever the Gnosis primitive command `dump` is executed as part of a run; due to this convention, for any given Gnosis model `model.gn`, the corresponding file `model.csv` is referred to as the *dump file*. The techniques in this paper are concerned with the data about traces and system state that are found in `model.tr` and `model.csv`, as highlighted in yellow in Figure 1.

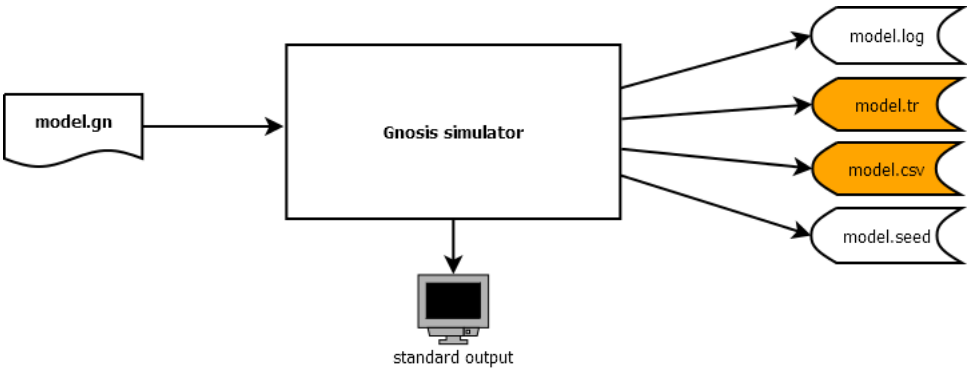

Figure 1: Performing a simulation of a model using Gnosis.

1.3 Our Contribution

We have:

- found a way to reason efficiently about properties of the state in a Gnosis simulation model. This facilitates reasoning about real-world security problems by extending existing work on the Gnosis simulation tool at HP Labs;
- produced a method to extract meaningful state information from multiple simulation runs of a Gnosis model;
- implemented a Python program which demonstrates the approach on a sample dump file, for a single run of a simulation;
- provided a means of specifying, in logical form, properties of state that we are interested in, so that simulation outputs can be filtered accordingly, and
- developed an algorithm for converting the selected (filtered) state changes into a finite state automaton, which is a formal abstract model that captures the essential properties of the original model under consideration.

1.4 Related and Previous Work

The results we have obtained in this report in the context of the Gnosis simulation framework are novel, and to our knowledge there is no directly comparable implementation. However, we are conscious of the relevance and linkages between our work and related work on automata chains and process mining. We are also conscious of the fact that our ideas require basic knowledge of automata theory, process algebra, formal verification and model checking in particular. For a standard reference refer to [5, 1].

In the context of resource-based logics such as SCRP (Synchronous Calculus of Resource and Process [6, 7]), Matthew Collinson developed some preliminary tools for basic resource-oriented model checking of Core Gnosis.

1.4.1 Automata Chains

The authors Grastien, Cordier, Largouët [8] have devised a formal theory of automata chaining, namely, ways to join together automata with common states. This is related, but much more extensive, to our method of joining together the automata extracted from different simulation runs. In our approach, we match state labels from different runs and only identify them if their labels are identical; an area for future work is to investigate more sophisticated ways of merging automata together, particularly for special or corner cases where additional transitions (often just ε-transitions) are needed to merge more complex automata.

1.4.2 Process Mining

After the work described in this report was completed, we became aware of related work by Van der Aalst and others on *process mining* [10]. The key difference between our approach and that advocated by the creators of process mining lies in their preference for the use of Petri net models rather than finite-state automata. However, we note that it is possible to convert Petri net models to automata, and that in [10] a tool is described that can be used to model-check LTL formulae over process models extracted from logs. Certainly the objectives of process mining are very much in alignment with the ideas presented here, and we envisage carrying over relevant ideas into our work.

2 The Extraction Algorithm

We have conceived a method of analysing essential properties of multiple runs of a simulation model, particularly in the context of the Gnosis modelling and simulation framework. Our method extracts information from the output of each simulation run, namely information about state changes, and constructs a graph corresponding to a finite state automaton with a simple state transition function. The

graphs from multiple runs can be combined into one bigger automaton, and this latter construction is a particularly useful representation of characteristic behaviours in the original model. The automata produced by the procedure are amenable to subsequent analysis and reasoning via model-checking techniques.

Our method presupposes the existence of:

- a simulation model M (which includes statements that produce observable outputs when run)
- a list $L = v_1, v_2, \ldots, v_k$ of "watch variables" (which are variables in the model whose state changes produce observable output)
- a simulator (in our experiments we have been focusing on the Gnosis tool) and a means of running it repeatedly on the simulation model to produce dump files.

The method itself only makes use of the output of Gnosis when the model M is simulated; in particular our solution just processes the dump files produced. Let $D = d_1, d_2, \ldots, d_n$ denote the set of dump files[2] generated by running Gnosis n times on model M.

2.1 Algorithm for Extracting State Information from Gnosis Simulations

Each run of the Gnosis simulator on an input model `model.gn` will produce two files of interest, the trace file `model.tr` and corresponding dump file `model.csv`. In order to extract meaningful information from a run of `model.gn`, we need to link together the information found in these two files; in particular, we need to know, for each step in the trace (or at least specified steps in the trace; where each step assumed to appear as a single printed line), what the corresponding values are of all the variables in the model — namely, the system state.

A naïve approach to linking trace information with the states of variables at different points during execution would be to include explicit `print` statements outputting the values of variables in different parts of the original model `model.gn`. However, our approach here is to extract information from simulation runs without modifying the source model.

In order to link the state information contained in a dump file with the trace information in a corresponding trace file we use Algorithm 2.1. What this algorithm does is look for strings in the trace file (specifically, sentinel characters that we have defined in advance or simply a string such as "Dumping State..."). Whenever such a string is encountered we know that the current system state is in the next line of `model.csv` (such is the functionality provided by the `dump;` statement in

[2] In the case of Gnosis, in order to run the algorithms presented in this section we need to make use of both *execution traces* and *dump files*, the latter being listings of the states of variables in a model at specified time instants in a particular run.

Gnosis) so we pick up the current system state from line d_{ind} of the dump file and increment the line counter *ind*.

The output of Algorithm 2.1 is a sequence of pairs of lines from the trace file `model.tr`, and matching system state printouts from `model.csv`. This functionality is required for the main algorithm in this report, which uses the system state to filter out parts of simulation traces.

Algorithm 1 The algorithm for linking traces and dump files produced by Gnosis during simulation runs.

Set $ind \leftarrow 1$
for all lines/transitions t_i in a trace file T produced by Gnosis during a run **do**
  if the dump marker/sentinel is found in t_i **then**
    Match t_i with line d_{ind} of the dump file D; Output (t_i, d_{ind}).
    Set $ind \leftarrow ind + 1$
  else
    Ignore line t_i of trace
  end if
end for

2.2 Processing A Single Simulation Run

The algorithm in this section extracts state changes from a single simulation run and produces a graph consisting of these changes. The procedure is simple, and it is the crux of the main algorithm in section 2.3. In Appendix C.1 an example execution of the algorithm is shown.

The method processes each trace t_i where $(1 \leq i \leq n)$, as described in Figure 2.2.1.

Algorithm 2 `process_single_run`(t_i, W): The algorithm for processing a single simulation run.

for all lines in the file t_i **do**
  Initialize, for all i, $currentstate[v_i] \leftarrow state[v_0]$.
  for all watch variables $v_1, \ldots, v_i, \ldots, v_k \in W$ **do**
    if $state[v_i] \neq currentstate[v_i]$ **then**
      Add transition $currentstate[v_i] \Rightarrow state[v_i]$ to graph/automaton
      $currentstate[v_i] \leftarrow state[v_i]$
    end if
  end for
end for
Assemble the list of all state changes and construct a directed graph G_i whose nodes are labelled by the values of $v_1, v_2, \ldots, v_k$, and whose edges correspond to transitions.

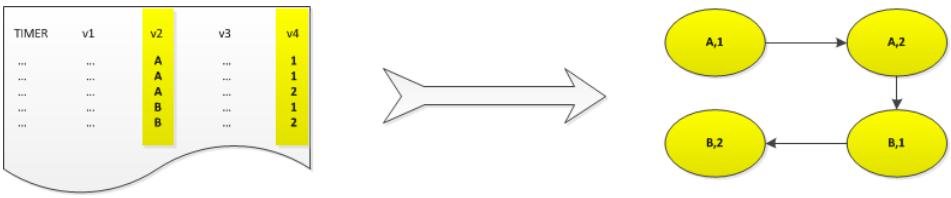

Figure 2: Processing a trace from a single simulation run to generate an automaton. In the example, the set of watch variables is {v2,v4} and no filter/formula has been applied.

Lines 50-69 of the Python program in Appendix A implement the algorithm in Figure 2.2.1. Figure 2 shows how this algorithm would work on a sample simulation run, where the highlighted columns correspond to watch variables (in this case named `v2` and `v4`). By identifying the state changes in these columns, the algorithm produces the graph shown in the right hand side of the figure. This graph is the first main result produced by our method, and the intention is that, with the refinements presented in the following sections, it provides a useful abstraction of the behaviour described by a simulation run.

2.2.1 Filtering State Changes of Interest

We have implemented a generalisation of the algorithm in Figure which identifies only a *selection* of the state changes found in a trace. This is very important for practical applications, as not all state changes are created equal: for many types of analysis, only certain state changes may be meaningful. By writing propositional formulas over the watch variables in a simulation model, it is possible to specify which state changes should be output. We have implemented this functionality in the program in Appendix A - see lines 35-44 and 56, which invokes the formula evaluation procedure on each line of a trace. The formula evaluation procedure is shown in lines 128-138, and it invokes the other sub-procedures `parseformula` and `evalformula` as appropriate.

Note that despite the similarity with explicit-state model checking, that is not what we are doing here: in fact, as we shall see, we are interested in performing model-checking of formulas on the graph that is produced from the algorithm in Figure 2.2.1, but only *after* we have filtered the state changes of interest. We are evaluating formulas on the states of a simulation run only for the purpose of determining whether they should be included in the final, extracted automaton. It is on *that* automaton that we would want to check formulae that express security or correctness requirements, as in model checking.

The program in Appendix A demonstrates the algorithm for a realistic trace file (this file is similar to a real Gnosis trace file, and is intended for demonstration purposes). The script also demonstrates the ability to filter lines from d_i, so that only lines satisfying a logical condition are used in constructing the graph. This is a first example of reasoning over the state changes in a simulation model. See Figure 2 for a diagram demonstrating the approach for a single simulation run.

2.2.2 Example of filtering:

If the watch variables for a given model are v_2, v_4, which can take values in `A`, `B` and $1, \ldots, 10$ respectively, we could specify that the graph should be constructed only by considering lines where $v_2 =$ `A` and $v_4 > 1$. For the demonstration script in Appendix A we would give the formula `v2=A,v4>1` as input in this case (along with a suitable dump file), where the comma (,) indicates conjunction of formulae.

The method is generalised to the processing of multiple runs of a simulation

model by processing each dump file as above and then unifying the graphs/automata produced in each case together, producing one large automaton. It is this large automaton that may be deemed to characterize essential features of the original model, and we envisage checking logical formulae on this automaton (model checking).

2.3 Main Algorithm: Processing Multiple Simulation Runs

Figure 3 shows a simple algorithm for merging the automata corresponding to several different runs into a single automaton. The algorithm in its presented form is almost trivial, as all it does is detect edges appearing in both graphs G and g_i (by comparing the states in s and e for equality) and adds the edges that are in g_i but are missing from G. Line 6 of this algorithm is where a more sophisticated comparison could be introduced in future work.

Definition 1 (Emergent Structure) *We will use the term* emergent structure *to refer to the mathematical object generated (a finite-state automaton) by performing the extraction algorithm in Figure 3 to multiple runs of a given simulation model.*

Algorithm 3 `process_multiple_runs`(T,W): The algorithm for combining the automata/graphs from multiple simulation runs with traces $t_i \in T$ and watch variables $v \in W$.

1: $G \leftarrow \emptyset$
2: **for all** simulation runs with respective traces t_i **do**
3: $g_i \leftarrow$ `process_single_run`(t_i,W) where W is the set of watch variables (see Figure 2.2.1).
4: **for all** edges (s,s') in G **do**
5: **for all** edges (e,e') in g_i **do**
6: **if** $(s = e)$ **then**
7: Do nothing; keep edge as is
8: **else if** $(s \neq e)$ **then**
9: Add edge (e,e') to G
10: **end if**
11: **end for**
12: **end for**
13: **end for**

2.3.1 Computing Probabilities of Particular Subtraces

The main algorithm we have discussed will produce the structure of a graph identifying state transitions that occur in multiple runs of a simulation; as we have seen, we can control which state transitions are to be included in the graph by selecting watch variables and applying filters on the values of variables. We can go further, and annotate state transitions with the probability of their occurrence, using the simple frequency-based definition of probability. To do this, we introduce a counter into Algorithm 3 which is incremented each time a particular state transition is found in a simulation run/trace. We compute the probability using the following definition:

$$P_{a\to b} = \frac{\text{weight of state transition } a \to b \text{ in current trace}}{\text{total number of transitions from state } a \text{ to other states}} \tag{1}$$

The variables a, b range over all states in the graph/automaton being generated. We require that for all values of a, b, the sum of all probabilities is unity:

$$\sum_{a,b} P_{a\to b} = 1 \tag{2}$$

Detailed Example of Probability Calculation. Consider a simulation that produces three different runs with the following traces. The simulation has three designated states denoted `PROCESSING`, `SANITIZING`, `COMPLETE` (which are assumed to correspond to changes in some watch variables of interest).

Simulation Run R_1	**Simulation Run R_2**	**Simulation Run R_3**
PROCESSING →SANITIZING	PROCESSING →SANITIZING	PROCESSING →SANITIZING
SANITIZING →COMPLETE	SANITIZING →PROCESSING	SANITIZING →PROCESSING
	PROCESSING →SANITIZING	PROCESSING →SANITIZING
	SANITIZING →COMPLETE	SANITIZING →PROCESSING
		SANITIZING →COMPLETE

Here are the frequencies of occurrence (the *weights*) of the different transitions in each run:

Transition \ **Run**	R_1	R_2	R_3
PROCESSING→SANITIZING	1	2	2
SANITIZING→PROCESSING	0	1	2
SANITIZING →COMPLETE	1	1	1

Ignoring the order in which transitions occur in each run, we can compute the probability that a transition from state x to state y will occur (where $x, y \in \{A, B, C\}$ in this example) using equation 1:

- In run R_1:

  Transition PROCESSING→SANITIZING occurs with probability $\frac{1}{1} = 1$.
  Transition SANITIZING →COMPLETE occurs with probability $\frac{1}{1} = 1$.
  Resulting graph:

- In run R_2:

  Transition PROCESSING→SANITIZING occurs with probability $\frac{2}{2} = 1$.
  Transition SANITIZING→PROCESSING occurs with probability $\frac{1}{2}$.
  Transition SANITIZING →COMPLETE occurs with probability $\frac{1}{2}$.
  Resulting graph:

- In run R_3:

  Transition PROCESSING→SANITIZING occurs with probability $\frac{2}{2} = 1$.
  Transition SANITIZING→PROCESSING occurs with probability $\frac{2}{3}$.
  Transition SANITIZING →COMPLETE occurs with probability $\frac{1}{3}$.
  Resulting graph:

The final metric we use as a probability for each of the transitions in all simulation runs is the average of the probabilities of the particular transition in all runs. If we denote by $p^{avg}_{x\to y}$ this metric we get, for the above example:

$$\begin{aligned} p^{avg}_{\texttt{PROCESSING}\to\texttt{SANITIZING}} &= \frac{1+1+1}{3} = 1 \\ p^{avg}_{\texttt{SANITIZING}\to\texttt{PROCESSING}} &= \frac{0+\frac{1}{2}+\frac{2}{3}}{3} = \frac{7}{18} \\ p^{avg}_{\texttt{SANITIZING}\to\texttt{COMPLETE}} &= \frac{1+\frac{1}{2}+\frac{1}{3}}{3} = \frac{11}{18} \end{aligned}$$

The final output of the algorithm, once the above values have been computed, will be this graph:

3 Discussion and Directions for Future Work

The advantages of this approach are manifold. First, Gnosis simulation models tend to be extremely large and complex. Identifying entire regions of Gnosis models with a single state descriptor can be extremely useful in practice, as it allows the user to abstract away from many details; in fact, the finite-state automata generated by our technique can be easily visualised and this would surely assist in the human understanding of any Gnosis model.

So the first great advantage of our method is the conciseness it provides in relation to existing techniques for understanding Gnosis models. The most important advantage is that the method paves the way for model checking of Gnosis models, or parts thereof. Model checking is a well-established means of gaining assurance and confidence in the correctness of systems, and most importantly it is useful for automatically detecting conceptual flaws or errors in system designs. We note here that in order for an analysis using our method to be truly beneficial, the initial Gnosis simulation model has to be a very accurate representation of the real-world problem under consideration. Our method creates an abstraction of an existing model, so there is always the danger of obtaining overly general conclusions or results that are divorced from important aspects of the real-world system or problem under consideration. Despite the above caveat, we believe that our method of analysis can bring significant benefits if combined with a suitable model-checking algorithm and a suitable logic for specifying properties of multiple runs of models.

We have identified a number of directions for future work, including those listed below.

Model checking of security and correctness properties of emergent structure. One of the primary motivations for the algorithms presented in this paper is the need to be able to formally and automatically check correctness and security properties of systems being simulated. Simulation of system behaviour can yield useful insights, but it is never exhaustive; *model checking*, on the other hand, consists of systematically exploring all possible behaviours of a given model. By extracting the emergent structure of a simulation, we can systematically explore its behaviour (as the emergent structure is much smaller than the original model) and identify potential issues, bugs, unexpected possibilities. The automata produced by the algorithms presented here are discrete, probabilistic finite state automata and so could be fed into probabilistic model checking tools such as PRISM [9]. While we have not discussed time explicitly in this paper, there are many simulation models for Gnosis in which time plays an important rôle, and so we foresee potential linkages to timed automata [2], model checking of LTL and CTL properties [5], as well as use of tools such as UPPAAL and KRONOS (see [3] for a discussion of these tools) for analysis. Statistical model checking [4] also looks like a promising direction to consider.

Reconstruction of simulation models from emergent structure. A fundamental assumption of the algorithms presented in this paper has been that the user has little or no knowledge of the structure of the original models being analysed. It is conceivable that, by processing the emergent structure corresponding to a given set of simulations, one could try to *reconstruct* and/or *approximate* the original simulation model. This can be seen as a form of reverse engineering, and developing tools to do this may well be useful for security applications.

Derivation of structural models from statistical data and discovery of *process patterns* and symmetries. The intention of our work has been to extract structure from outputs of simulations that are related; the emergent structure is supposed to provide insight into the behaviour expressed by a complex simulation model. Investigating further how to extract process patterns from system traces is an essential part of this work. Furthermore, when the original model contains much statistical data, we may be interested in extracting only the overall structure and ignoring as much statistical noise as possible. Clearly, this is an important area for future work.

Comparison of related simulation models for trace refinement. Clearly, the emergent structures of different simulation models can be compared directly; rather than just checking for equality, one could compare emergent structures for relations of refinement and containment. It could be that some traces of one model appear in another when this is not expected at all. We believe that this is a particularly important issue to investigate in security applications, where particular types of attacker present specific, but repeatable patterns of behaviour.

Separation of system/environment behaviours from attacker descriptions and assumptions. In current work on security analytics with Gnosis, it is common for a single simulation model to contain descriptions of a system, its environment, and a particular attacker "all in one." By extracting emergent structure from models that contain only system behaviour, we can consider different attacker models in a very modular fashion.

We envisage a number of applications of these techniques in the context of security analytics and in the analysis of security properties of complex, concurrent system models. Here are some of these:

- **Analysis of security logs of cloud infrastructures**
- **Detection of patterns of malicious behaviour in system behaviour**
- **Comparison of attacker models**

4 Conclusions

In this paper we have presented a method, and experimental implementation, for extracting emergent structure from system simulation models. We have documented algorithms for extracting state changes from individual simulation runs, and combining these state changes together when processing multiple runs. The purpose of this work is to obtain meaningful representations of complex simulation models and we envisage many different practical applications in future work.

For the purposes of this paper, we have focused on the Gnosis simulation framework, which is used extensively for security analytics modelling at HP Labs. While formal verification tools have not been previously developed for use in conjunction with Gnosis, we have developed algorithms and a prototype that serves as a starting point for explicit-state model checking of Gnosis simulations. We believe this is a very promising avenue of research, with important practical applications and exploitation routes.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098, pages 87–124. Springer-Verlag, 2004.

[3] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[4] E. M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *ATVA 2011: 9th International Symposium on Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 1–12, 2011.

[5] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[6] Matthew Collinson, Brian Monahan, and David Pym. Semantics for structured systems modelling and simulation. In *Proceedings of Simutools 2010*. ACM Digital Library, 2010. ISBN: 78-963-9799-87-5.

[7] Matthew Collinson, Brian Monahan, and David Pym. *A Discipline of Mathematical Systems Modelling*. College Publications, 2012.

[8] Alban Grastien, Marie-Odile Cordier, and Christine Largouët. Incremental diagnosis of discrete-event systems. In *Sixteenth International Workshop on Principles of Diagnosis (DX-05)*, 2005.

[9] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[10] Wil M. P. Van Der Aalst. *Process Mining*. Springer Verlag, 2011.

A Python Source Code for the Processor

```
1  # Program to process output of Gnosis and generate automaton
       from a single run
2  # by looking only at dump file
3
4  # Author: Nick Papanikolaou, HP Labs
5  # (Based on joint research with Brian Quentin Monahan)
6
7  # use file "fakedumpfile.txt" for initial example
8  # Interesting cases:
9  # st1,st2 with st1=A,var2=3
10 # st1,st2 with var2=2
11
12 # rhs could be a variable!! next step
13 # general reasoning about logs
14 # security event logs
15
16 import fileinput
17 import string
18
19 def processdumpfile(f) :
20     # Input dump file and split into a list of lines
21     cont=''
22     for line in f:
23         if (len(string.strip(line))>0) and (line[0]!='#'):
24             cont = cont + line
25
26     all_lines = cont.splitlines()
27
28     # Input comma-separated list of watch variables (state
           descriptors)
29     varlist = raw_input("Enter comma-separated list of
           variable names to follow: ").split(",")
30
31     # Pick up actual variables listed in dump file, and find
           columns needed
32     indicesofcols = locatecolumns(varlist,all_lines)
33
34     #NEW: Filtering - input a formula
35     formula = raw_input("Enter comma-separated list of
           formulae (comma is conjunction): ")
36
```

```
37      all_formulas = formula.split(",")
38      listofformulastoevaluate = []
39      if len(formula)!=0:
40          for f in all_formulas:
41              (indexofvar, op, rhs) = parseformula(f,all_lines)
42              listofformulastoevaluate.append((indexofvar,op,rhs))
43      else:
44          print "No formula entered."
45
46      # Examine the columns corresponding to watch variables and
            find state changes
47      saved_state = []
48      state_changes_list = []
49      transition = ''
50      for line in all_lines:
51          curr_line_fields = line.split("\t")
52          # Check if this line satisfies all formulae
53          if len(listofformulastoevaluate)==0:
54              filter_holds = True
55          else:
56              filter_holds =
                    multiformulaevaluate(listofformulastoevaluate,
                    curr_line_fields)
57          if (filter_holds):
58              curr_state=[]
59              for i in indicesofcols:
60                  curr_state = curr_state + [curr_line_fields[i]]
61              if ((saved_state==[]) or (saved_state==varlist) or
                    (curr_state==varlist)):
62                  saved_state = curr_state
63              elif (curr_state != saved_state):
64                  transition = str(saved_state) + ' -> ' +
                        str(curr_state)
65                  print transition
66                  state_changes_list = state_changes_list +
                        [transition]
67                  saved_state = curr_state
68              else:
69                  print "(no state change)"
70
71      # Look through list of state changes and count them to
            compute edge weights
72      newlist = []
73      for el in state_changes_list:
```

```
74          weight = 0
75          for el2 in state_changes_list:
76              if (el==el2):
77                  weight = weight + 1
78          newlist = newlist + [(el, weight)]
79          weight = 0
80
81      # Print final list of state changes with weights
82      print "List of state changes corresponding to variables "
            + str(varlist) + " is as follows:"
83      unique = list(set(newlist))
84      for t in unique:
85          print t[0], "with weight ", t[1]
86
87  def locatecolumns(varlist, inputlines):
88      columnnames = inputlines[0].split("\t")
89      indicesofcols = []
90      currindex = 0
91      for var in varlist:
92          currindex=0
93          while (currindex < len(columnnames)):
94              if (columnnames[currindex]==var):
95                  indicesofcols.append(int(currindex))
96                  currindex = currindex + 1
97              else:
98                  currindex = currindex + 1
99      return indicesofcols
100
101 def evalformula(operator,rhs,value):
102     if (operator=='<'):
103         return (value < rhs)
104     elif (operator=='>'):
105         return (value > rhs)
106     elif (operator=='='):
107         return (value == rhs)
108     else:
109         print "Error evaluating formula!"
110
111 def parseformula(formula, inputlines):
112     ## tokenize formula and print out its parts
113     ops = ['>','<','=']
114     formulaparts = []
115     for op in ops:
116         formulaparts = formula.split(op)
```

```
117         if len(formulaparts)>1:
118             formulaparts = formulaparts + [op]
119             break
120     lhs = formulaparts[0]
121     op = formulaparts[2]
122     rhs = formulaparts[1]
123 
124     indexofvar = (locatecolumns([lhs], inputlines))[0]
125     print "Variable in the formula is ", lhs, " and its
            column number is", indexofvar
126     return (indexofvar, op, rhs)
127 
128 def multiformulaevaluate(formulaelist, linefields):
129     result = True
130     for formula in formulaelist:
131         col = formula[0]
132         op = formula[1]
133         rhs = formula[2]
134         e = evalformula(op,rhs,linefields[col])
135         #print "Evaluating formula", op, rhs, "on column",
                col, "with value", linefields[col], " RESULT = ", e
136         #comma corresponds to conjunction of formulae
137         result = result and e
138     return result
139 
140 def main():
141     dumpfile = open("fakedumpfile.txt")
142     processdumpfile(dumpfile)
143 
144 # DONE (WEIGHT) Add code above for adding weights to
        transitions and then more for computing probability of a
        transition!
145 # DONE Input should be list of variables to watch (more than
        one!)
146 # DONE Evaluate formulae on each line
147 # TODO Richer formulae - disjunction (;) and brackets
148 main()
```

B Contents of Example Input File

```
# An example of what a trace might look like

TIMER    ID       var1     var2     st1      st2
0.00     1        1        1        A        X
0.35     1        1        3        A        X
0.46     1        1        3        A        Y
1.00     1        1        3        A        X
1.02     1        1        3        A        Y
1.10     1        1        2        B        Y
1.15     1        1        2        B        Y
1.20     2        1        2        B        Y
1.25     2        1        2        B        Y
1.46     2        1        2        B        X
1.59     2        1        2        A        Y
13.0     2        2        4        A        X
14.0     2        2        3        C        Z

# end of example trace file
```

C Sample Runs of the Processor

C.1 Running The Processor With Two Watch Variables

Below is the listing of the processor's outputs when given the input file in Section B.

```
1  Enter comma-separated list of variable names to follow:
       var2,st2
2  Enter comma-separated list of formulae (comma is conjunction):
3  No formula entered.
4  ['1', 'X'] -> ['3', 'X']
5  ['3', 'X'] -> ['3', 'Y']
6  ['3', 'Y'] -> ['3', 'X']
7  ['3', 'X'] -> ['3', 'Y']
8  ['3', 'Y'] -> ['2', 'Y']
9  (no state change)
10 (no state change)
11 (no state change)
12 ['2', 'Y'] -> ['2', 'X']
13 ['2', 'X'] -> ['2', 'Y']
14 ['2', 'Y'] -> ['4', 'X']
15 ['4', 'X'] -> ['3', 'Z']
16 List of state changes corresponding to variables ['var2',
       'st2'] is as follows:
17 ['2', 'X'] -> ['2', 'Y'] with weight 1
18 ['2', 'Y'] -> ['2', 'X'] with weight 1
19 ['2', 'Y'] -> ['4', 'X'] with weight 1
20 ['3', 'Y'] -> ['3', 'X'] with weight 1
21 ['3', 'Y'] -> ['2', 'Y'] with weight 1
22 ['1', 'X'] -> ['3', 'X'] with weight 1
23 ['3', 'X'] -> ['3', 'Y'] with weight 2
24 ['4', 'X'] -> ['3', 'Z'] with weight 1
```

C.2 Running The Processor With Two Watch Variables And A State Filter

Below is the listing of the processor's outputs when given the input file in Section B.

```
1  Enter comma-separated list of variable names to follow:
       var2,st2
2  Enter comma-separated list of formulae (comma is
       conjunction): var2>2
3  Variable in the formula is var2 and its column number is 3
```

```
['3', 'X'] -> ['3', 'Y']
['3', 'Y'] -> ['3', 'X']
['3', 'X'] -> ['3', 'Y']
['3', 'Y'] -> ['4', 'X']
['4', 'X'] -> ['3', 'Z']
List of state changes corresponding to variables ['var2',
    'st2'] is as follows:
['3', 'X'] -> ['3', 'Y'] with weight 2
['4', 'X'] -> ['3', 'Z'] with weight 1
['3', 'Y'] -> ['3', 'X'] with weight 1
['3', 'Y'] -> ['4', 'X'] with weight 1
```