Journal of Simulation (2010) 4, 170–180

www.palgrave-journals.com/jos/

INSIGHT: understanding unexpected behaviours in agent-based simulations

R Gore* and PF Reynolds Jr

University of Virginia, VA, USA

Unexpected behaviours in simulations require explanation, so that decision-makers and subject matter experts can separate valid behaviours from design or coding errors. Validation of unexpected behaviours requires accumulation of insight into the behaviour and the conditions under which it arises. Agent-based simulations are known for unexpected behaviours that emerge as the simulation executes. To facilitate user exploration, analysis, understanding and insight of unexpected behaviours, we have developed a novel semi-automated methodology, INSIGHT. INSIGHT provides: (1) semi-automatic hypothesis testing for exploring an unexpected behaviour, and (2) automatic identification of statements in an agent-based simulation's source code which have the strongest influence on an unexpected behaviour. INSIGHT is applicable to both deterministic and stochastic agent-based simulations and extends the state of the art in agent-based simulation analysis.
Journal of Simulation (2010) **4,** 170–180. doi:10.1057/jos.2009.26; published online 27 November 2009

Keywords: simulation; methodology; computational analysis; validation; agent-based modelling; emergent behaviour

1. Introduction

Simulations have entered the mainstream of critical public policy and research decision-making practices (Whipple, 1996; Arthur, 1999; Hooke and Pielke, 2000; Cha, 2005; Elderd *et al*, 2006; National Science Foundation, 2006). Public policy-makers and scientists look to simulation for insight, trends and likely outcomes. Unfortunately, methods for gaining insight into unexpected outcomes, as related to model design and simulation implementation and use, have not kept pace. This problem is magnified in agent-based simulations where unexpected behaviours typically *emerge* as the simulation executes. *Emergent behaviours* in agent-based simulations represent the arising of novel structures, patterns or properties during the process of self-organization in a complex system (Corning, 2002). These emergent behaviours may be either expected by the simulation user or unexpected. Unexpected emergent behaviours often occur in agent-based simulations where the simulation specification is incomplete because the application domain is poorly understood; the purpose of the simulation is to explore the domain of interest (Trenouth, 1991). Writing the agent-based simulation becomes a theory construction task where the simulation is the expression of the theory (Wielinga, 1978). Trusted simulations, agent-based or not, in the same application domain, data sets from physical experiments, and subject matter expert opinions are used to test the theory. This is what gives exploratory agent-based simulations their experimental nature. Those emergent behaviours that are not defined in the specification and do not match the behaviour of other trusted simulations, data sets from physical experiments or subject matter expert opinions are *unexpected behaviours*. These unexpected behaviours require understanding and explanation to determine if the behaviour is an error or new knowledge in the application domain.

Uncertainty about unexpected agent-based simulation behaviours has fuelled recent public policy debate (Cha, 2005). A case in point is the agent-based disease spread simulation, Episims, which simulates nationwide spread of the smallpox virus under various vaccination strategies (Eubank *et al*, 2004). Previous studies of smallpox concluded that a targeted vaccination strategy could manage disease spread as well as mass vaccination of the entire population. However, the Episims studies showed that disease spread under a targeted vaccination strategy is much more severe than under a mass vaccination strategy. Differences in these predictions led to policy debate over 'whether or not it's necessary to synthesize enough smallpox vaccine for the entire country' (Cha, 2005).

The Institute of Medicine of the National Academies has published a collection of critical opinions of the simulation predictions of Ep isims. The chief complaint raised is that the simulation developers could not provide a clear explanation for the difference between their vaccination strategy predictions and those previously established (Baciu *et al*, 2005). Given the increasing use of agent-based simulations in critical decisions, a methodology that facilitates understanding and establishment of validity of unexpected behaviours is needed.

*Correspondence: R Gore, 136 Hessian Hills Circle, Apt. 1, Charlottesville, VA 22901, USA.
E-mail: rjg7v@virginia.edu

We present a novel methodology, INSIGHT, that allows users to understand and validate or reject unexpected agent-based simulation behaviours efficiently and with confidence. We note an important difference between validating a simulation and validating an unexpected behaviour that arises in a simulation. The former represents an effort to demonstrate expected behaviour (Balci, 1997). The latter is a demonstration of the validity of behaviour that was unexpected for a given set of conditions, or experimental frames (Zeigler *et al*, 2000). Validation of unexpected behaviour requires accumulation of insight into, and understanding of, the behavior and the conditions under which it arises. Then the problem, the model for it, and any related simulation, are all reframed so that the unexpected behaviour either becomes a part of a set of behaviours one considers valid, or it is deemed invalid. INSIGHT combines semi-automated hypothesis testing (SAHT), program slicing, causal analysis and generation of program slice distribution functions to define an end-to-end automated process for discovering, analysing and understanding sources of unexpected behaviours in agent-based simulations. Table 1 itemizes where these technologies appear in the INSIGHT process and the outcomes of each.

The first component of INSIGHT is SAHT. SAHT allows a simulation user to observe characteristics of an unexpected behaviour as a simulated phenomenon is driven towards conditions of interest. Owing to the complexity of agent-based simulations where unexpected behaviours frequently occur, the user often does not know how to drive the simulation to conditions of interest directly. The term *condition of interest* means a simulated state, or set of states, in which a simulated phenomenon is maximized, minimized or targeted to an exact requirement (Gore *et al*, 2007).

Causal Program Slicing (CPS) combines program slicing and causal inference, in a novel manner, to provide insight into the interactions of simulation variables and source code statements that cause unexpected behaviour. CPS precision suffers when a simulation includes stochastics, as most do. As a remedy, we employ Program Slice Distribution Functions (PSDFs) to quantify the uncertainty of dynamic program slices. Applying PSDFs to CPS increases precision in the CPS analysis for stochastic simulations.

Use of INSIGHT is not limited to agent-based simulations. However, it is particularly useful to agent-based simulations because of their frequent tendency to exhibit unexpected behaviours (often regarded as a feature of agent-based simulations) that turn out not to be easily attributed to specific blocks of code. INSIGHT is applicable to all simulations written in any high level programming language (eg Java, C#, C++, etc) and requires access to the simulation source code. INSIGHT remains in development and is unavailable for download; however, in the future we expect to release it through open source channels to the simulation community.

Table 1 Technologies comprising INSIGHT

Abbreviation	*Descriptive name*	*Function*	*Outcome for user*
SAHT	Semi-automated hypothesis testing	Employ Semi-automated model adaptation methods to force a simulation to exhibit user-defined conditions of interest.	Increased user confidence about the hypothesized conditions controlling the unexpected behaviour.
CPS	Causal program slicing	Apply program slicing and causal inference to produce quantified insight into relationships among simulation inputs, variables with the execution flow of the simulation and unexpected behaviours.	With focus guided by hypotheses reinforced by SAHT, user acquires CPS generated quantified causal execution flows affecting unexpected behaviours, thus increasing user insight into causes of those behaviours.
PSDF	Program slice distribution functions	Generate distributions related to the likelihood of different execution flows for specified simulation inputs.	In combination with CPS analysis user gains insight into relationship between quantified likelihood of execution flows and unexpected behaviour.
INSIGHT	Methodology for understanding unexpected behaviours in simulations	Semi-automatically increase insight into the causes of unexpected behaviours in a simulation.	Higher confidence explanations for unexpected behaviours, and analysis provided by INSIGHT. Confidence can often be quantified based on INSIGHT outputs.

In the sequel, we describe the particulars of the SAHT, CPS and PSDF methods comprising INSIGHT and we describe an agent-based SEIR epidemic simulation where INSIGHT has been employed for its intended purpose; namely to provide insight into unexpected agent-based simulation behaviours.

2. INSIGHT components

In this section we describe the component technologies that comprise INSIGHT and offer example applications of the components.

2.1. Semi-automated hypothesis testing (SAHT)

SAHT is a method for increasing insight into unexpected behaviours, and supporting validation of valid unexpected behaviours. SAHT can be used to increase confidence in hypothesized meanings of unexpected agent-based simulation behaviours. Simulation validation is not a goal, but can be an outcome of SAHT.

A number of approaches to automatic model adaptation contain ideas that contribute to SAHT. However, these solutions do not address all of the needs for efficiently testing user specified hypotheses regarding unexpected program behaviour. Some adaptation solutions are constrained to specific domains (Reiher *et al*, 2000; Chang and Karamcheti, 2001), while others operate only at the syntactic interface level (Gschwind, 2002; Haack *et al*, 2002; Brogi *et al,* 2003). SAHT includes both these capabilities allowing users to create conditions of interest and modify model interfaces as we discuss next.

The SAHT process. When testing a hypothesis about an unexpected behaviour, a user may wish to observe the unexpected behaviour under a specified set of target behaviours. However, when there are non-linearities in behaviours of an agent-based simulation, the user may not know how to adapt the simulation to achieve the desired target behaviours directly. We advocate the application of semi-automated model adaptation for efficient exploration of unexpected behaviour under specified conditions. When constructing an agent-based simulation, abstractions inevitably must be selected in order to reduce complexity, improve performance, or provide estimations for unknown information. We call those places where a user can choose among abstractions, *abstraction opportunities*. Carnahan has developed a language, FlexML, and supporting tools for a user to identify abstraction opportunities and alternatives for the model abstractions (Carnahan *et al*, 2007).

Given an unexpected behaviour, *B*, a user must establish if the unexpected behaviour *B* is valid or invalid. To better understand an unexpected behaviour, a user generally needs to formulate a hypothesis, *H*, about how unexpected behaviour *B* will be manifested under a condition of interest, *C*. Using this paradigm, the user needs to identify possible model abstraction alternatives to search to create *C*. The user can test hypothesis *H*, by observing the unexpected behaviour *B*, under condition of interest *C* (Gore *et al*, 2007).

The FlexML tools enable alternatives for each model abstraction to be reflected in the simulation source code as possible alternate bindings (Carnahan *et al*, 2007). With these alternate bindings, a model adaptation strategy employing optimization becomes possible. SAHT uses our optimization-based adaptation technology, COERCE (Reynolds *et al*, 2007), to test user hypothesis *H*, about an unexpected behaviour *B*, by efficiently creating user-specified conditions of interest *C*. A user gathers insight by observing an unexpected behaviour *B*, under the conditions of interest *C*. If the observed behaviour matches the user's hypothesis *H*, it passes the test, otherwise it fails (Gore *et al*, 2007).

Applying semi-automatic hypothesis testing to the Dunham simulation. Epidemics have been modelled mathematically for over a century (Diekmann and Heersterbeek, 2000). The well-established differential equation SEIR model of infectious disease spread is shown in Figure 1. It is described by the system of differential equations depicted in Figure 1 where p, q, λ, γ, ε, and u are positive parameters and S, E, I, and R denote the fractions of the population that are susceptible, exposed, infectious, and removed, respectively. Individuals are susceptible, then exposed (in the latent period), then infectious, then removed from the studied population. The birth rate and death rate are assumed to be equal and denoted by u. The transfer rates of the disease between the exposed state and the infectious state and the infectious state and the removed state are denoted by ε and γ, respectively. The rate of new infections is described by the non-linear term $\lambda I^p S^q$ (Li and Muldowney, 1995).

Differential equation models are only one approach to studying disease spread. In contrast, Dunham (Dunham, 2005) has published an agent-based SEIR epidemic simulation. The Dunham simulation predicts disease spread by modelling interactions on a 2-D torus. Infectious individuals infect susceptible individuals within a specified radius and spread their infection with a given probability.

The results of the differential equation SEIR model and the Dunham simulation for a population of size 100 match closely: standard deviations of corresponding curves differ by less than 2% of the population size. Dunham's simulation appears to produce valid predictions. Differences between it

$$
\begin{aligned}
S' &= -\lambda I^p S^q + u - uS \\
E' &= \lambda I^p S^q + u - uS \\
I' &= \varepsilon E - (\gamma + u)I \\
R' &= \gamma I - uR
\end{aligned}
$$

Figure 1 Established differential equation SEIR model (Li and Muldowney, 1995).

and accepted data or models should be investigated to determine the model's general validity. When we tested Dunham's simulation over a range of reasonable conditions an unexpected behaviour emerged. Changing only the population parameter from size 100 to size 1000 significantly changes the predictions for disease spread. As shown in Figure 2(a) Dunham's simulation predicts a shorter, heightened infectious period where no infected individuals remain after day 35. In contrast, the differential equation model shown in Figure 2(b) predicts a longer infectious period with infectious individuals still present at day 80. The standard deviations of the Dunham's simulation curves from the accepted differential equation model's curves are at least 11% of the population size for each curve. These standard deviations, compared to those of the population of size 100 are an order of magnitude greater. Based on examples and claims in Dunham (2005) and the results of the models for a population of size 100, we expected the Dunham's simulation to predict results similar to the differential equation model. The differences represent an unexpected behaviour.

To understand why Dunham's simulation predictions differ so significantly for a population of size 1000, and to determine if the behaviour is valid, we applied SAHT. Epidemic models often include parameter(s) which represent the rate of new infections. New infections occur in Dunham's simulation over time but there is no input parameter for infection rate, and we found none published. We instrumented Dunham's simulation to capture the rate of new infections and used this to create the condition of interest *C*: the rate of new infections is 8.0 per infected individual. Using *C* as a target behaviour, the hypothesis *H* was tested: when the rate of new infections is 8.0 in both models, their predictions will be similar (Gore and Reynolds, 2008).

Dunham's simulation was adapted by searching alternative bindings for six different abstraction opportunities to create the condition of interest *C*: new infection rate = 8.0. When the condition of interest, *C*, was achieved, Dunham's simulation predictions closely matched those of the differential equation based model. This is shown in Figure 3(a) and (b). The standard deviations of Dunham's simulation curves from the accepted differential equation model's curves are less than 2% of the population size for each curve. The hypothesis is correct: the predictions are similar when the rate of new infections is 8.0.

SAHT hypothesis testing does not directly enable users to understand *how* the rate of new infections is controlled in Dunham's simulation, only that it is important to the simulation's behaviour. To gain this insight, CPS and

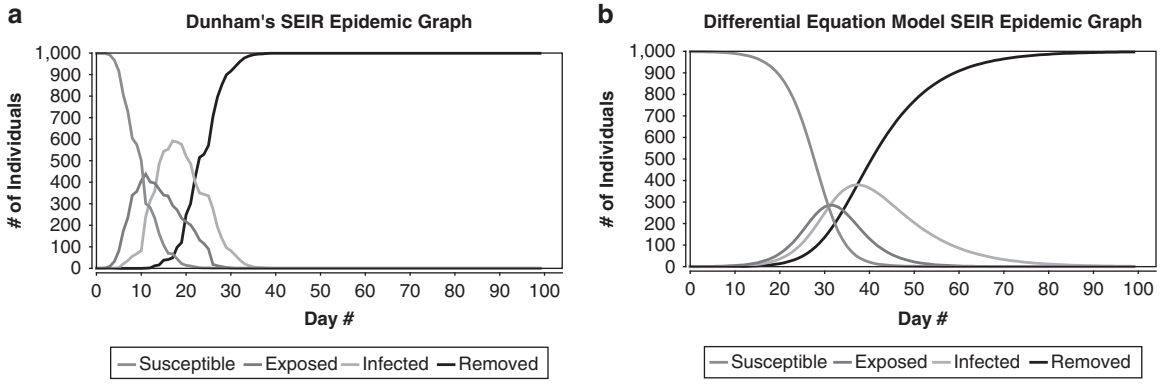

Figure 2 (a) and (b) Dunham's simulation and differential equation model for population of 1000 over 100 days.

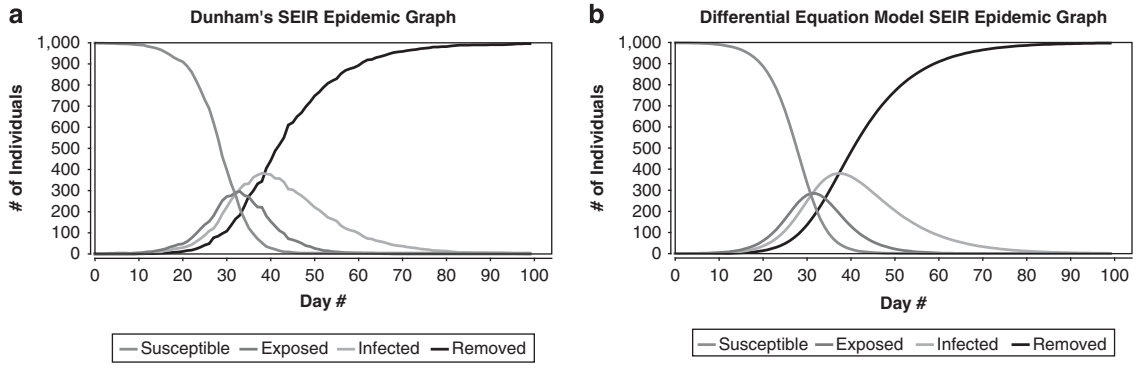

Figure 3 (a) and (b) Dunham's simulation under condition of interest *C* and differential equation model for population of 1000 over 100 days.

PSDFs are applied to the Dunham simulation as described in the following sections.

2.2. Causal program slicing (CPS)

CPS combines program slicing and causal inference to provide insight into the interactions of model variables and source code statements that cause an unexpected behaviour. Program slicing is a decomposition technique that extracts program statements relevant to a particular computation within the program (Weiser, 1984). Program slices are either static or dynamic; CPS uses only static program slicing. The criterion for a static slice is a 2-tuple consisting of $\{s, v\}$, where v is the variable of interest and s is the line number of the statement of interest. A static program slice contains no assumptions about the input to a program. The CPS analysis enabled by the use of static program slicing allows users to focus their attention on explaining *how* the identified program statements in the source code cause the unexpected behaviour.

Sensitivity analysis and design of experiments (DoE) have been proposed as experimental methods used to quantify indeterminate measurements of factors and interactions through observance of forced changes to explore the robustness of the behaviour in an agent-based simulation (Montgomery, 2004). These approaches have been refined to *data farming*. Data farming can provide users with insight into how agent-based simulation outputs vary in relation to agent-based simulation input parameters (Lucas *et al*, 2002). The goal of CPS is to move this analysis from the input-output level of the simulation to the source code level where simulation variables change state. This allows users to understand how the variations in the input parameters change the simulation variables in the source code and how the state changes in the simulation variables influence simulation behaviour. Performing this type of analysis via causal inference (Pearl, 2000; Spirtes *et al*, 2001) at the source code level of the agent-based simulation allows us to quantify the influence of a program statement on an unexpected behaviour.

The CPS process. CPS begins with user identification of the state of the simulation that represents the unexpected behaviour. The program statement in the simulation's source code at which this state can be observed is identified by its line number, s. The variable storing the value of interest related to the unexpected behaviour is identified by the variable, v. Static program slicing is then applied using the static slicing criterion $\{s, v\}$. The static program slice will identify all statements in the source code containing variables that may influence the state of the simulation representing the unexpected behaviour. The list of statements in the static program slice is passed to the CPS preprocessor.

CPS preprocesses simulation source code and inserts statements to capture state changes of variables in the static program slice. The inserted code collects the value of a variable in an identified program statement before and after the execution of the program statement. Thus state changes in program statement variables are mapped to simulation source code. The collected variables' values serve as samples which can be analysed to determine the influence a variable's state has on the unexpected behaviour or the state of another variable. Collection of values of variables as the variables change state throughout the simulation execution, and quantifying the influence the state changes in the variables have on the unexpected behaviour or on each other, is central to our work. Each of the quantified variable state changes can be mapped back to the program statement which caused the variable to change state. Further details on capturing the variable state for different types of program statements are provided in Gore and Reynolds (2009).

Once the preprocessing step is complete, the user next identifies the set of input parameters to explore. As a part of the CPS process, these parameters are varied to determine how changes in the parameters change the state of the variables in the simulation's source code. We employ sampling approaches with efficient and equal density coverage of the search space for a given number of samples. Examples of candidate sampling approaches include Latin Hypercube Sampling and Orthogonal Sampling (Loh, 1996; Garcia, 2000).

The simulation is executed for each input parameter configuration in the set. The code inserted by the CPS preprocessor collects the samples for each state of each variable in the static program slice as the simulation executes. Next, the influence of a variable state on unexpected behaviour or another variable state is quantified by applying causal inferencing to the samples of the variables' states. The result is a chain of variable states which specify how each variable state influences others, and the unexpected simulation behaviour. The strength of a causal influence is measured as the absolute value of the correlation coefficient (between [0, 1] inclusive) between two variable states. Using the correlation coefficient along with conditional independence to measure causality is based on previous causal inference research (Pearl, 2000; Spirtes *et al*, 2001).

The user identifies the threshold causal influence a variable state must have on the unexpected behaviour, or on another above threshold variable state influencing the unexpected behaviour to be included in the slice. The strength of a causal influence lies between $[0, 1]$ inclusive. Given a causal influence z, z has weak or no influence if $0.0 \leqslant z < 0.3$, moderate influence if $0.3 \leqslant z < 0.5$, and strong influence if $0.5 \leqslant z \leqslant 1$ (Cohen, 1988).

Using data stored by the preprocessor, each variable state with a causal influence that is over a user-specified threshold is mapped back to the program statement that caused the

variable's state to change. Finally, a graph of the chain of program statements that have a causal influence on the unexpected behaviour is displayed to the user. The graph is annotated with the causal influence each program statement has on unexpected program behaviour or another program statement over threshold. The graph focuses user attention on understanding those statements in the simulation's source code with the strongest causal influence on the unexpected behaviour.

Applying CPS to an example program. To elucidate the CPS process we apply it to the program in Figure 4(a). CPS proceeds as follows:

1. The user identifies the value of x in line 13 as the program state capturing the unexpected behaviour.
2. The user configures CPS to only collect those variable states within the same function that affect the value of x in line 13. The user also specifies a causal influence threshold of 0.5, this is the minimum influence a variable state must have on the value of x in line 13 or on another variable state having an influence $\geqslant 0.5$ on x in line 13.
3. The causal program slicer initiates static program slicing with the slicing criterion {13, x} to determine the program statements that may influence the computation of x in statement 13. These statements are shown in Figure 4(b).
4. For program statements in Figure 4(b) the preprocessor inserts code to map the state of the variables n, i, x and sum back to their respective program statements. The preprocessor code also collects the state of each variable.
5. The user performs orthogonal sampling to generate 1000 different values for input parameter n, and runs the program for each of the generated values. The values for n range between 1 and 10 000.
6. CPS performs causal inference on the generated samples. CPS outputs a causal graph including each variable state with an influence $\geqslant 0.5$ on the value of x in line 13 or on another variable state which has a causal influence $\geqslant 0.5$ on the value of x in line 13.
7. Each variable state in the causal graph is mapped back to the program statement that caused the variable to change state. A causal graph containing only the program statements with the strongest causal influence on the value of x in line 13 is output. This graph, which gives the user insight, is shown in Figure 4(c). The initial value of n, and the state of sum, where those integers $\leqslant$ n are added together have the strongest influence on the value of x in line 13.

Imprecise CPS analysis for stochastic simulations. For stochastic simulations, there exist different possible outputs (or behaviours) for a fixed input. CPS cannot offer precise analysis of stochastic simulations because CPS assumes that for each simulation input there exists only one possible output and that for each input there exists only one possible dynamic program slice to be executed. A dynamic slice extracts program statements relevant to a particular computation within the program for a specified input (Weiser, 1984). A dynamic program slicing criterion consists of {*input*, line number of statement *s*, name of variable *v*} (Tip, 1995). Precision is measured by the number of dynamic slices for a fixed input provided by the analysis divided by the number of possible dynamic slices for a fixed input (Van der Walt and Barnard, 2006). The first CPS assumption is never true for stochastic simulations and often the second CPS assumption is not true either.

Figure 5(a)–(c) elucidate the imprecision in CPS analysis of stochastic simulations. From the user's point of view the behaviour of the example program in Figure 5(a) is stochastic. Figure 5(b) and (c) show the two possible dynamic program slices using criterion {n = 13, 7, x} for the program in Figure 5(a). Figure 5(b) shows the dynamic

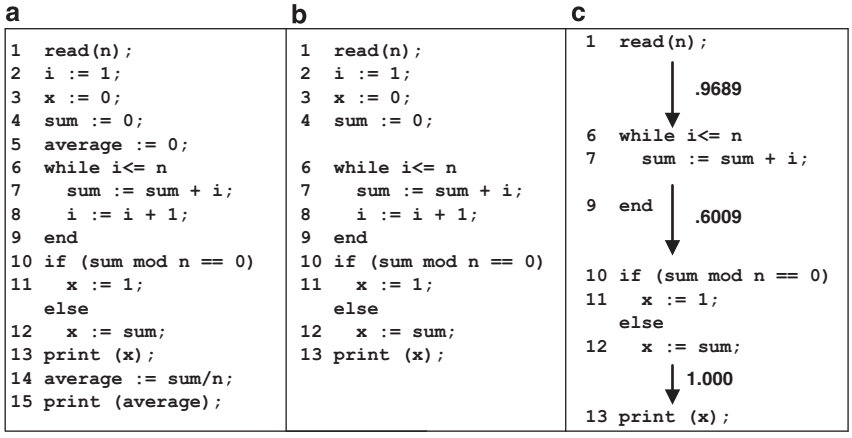

Figure 4 (a) A simple program; (b) a program slice using static slicing criterion {13, x}. (c) A causal program slice using criterion {13, x} and a threshold of 0.5.

a

```
1     read(n);
2     x = 0;
3     rand := randomNumber(0, 1);
4     if (rand >= .998 &&
          rand <= .999)
5        x := rand + n;
      else
6        x := n;
7     print (x);
```

b

```
1     read(n);
2     x = 0;

// (rand >= .998 &&
//  rand <= .999) == false


6        x := n;
7     print (x);
```

c

```
1     read(n);
2     x = 0;
3     rand := randomNumber(0, 1);
//    (rand >= .998 &&
//     rand <= .999) == true
5        x := rand + n;


7     print (x);
```

Figure 5 (a) A stochastic program; (b) one possible dynamic slice of the program using criterion {n = 13,7,x}. (c) Another possible dynamic slice of the program using the same criterion.

program slice, when the random number generator does not generate a number between 0.998 and 0.999. Figure 5(c) shows the dynamic program slice when the random number generator does generate a random number between 0.998 and 0.999. Assuming a uniform random number generator, the dynamic program slice for the program in Figure 5(a) is the one shown in Figure 5(b) approximately 99.9% of the time and the one shown in Figure 5(c) 0.1% of the time.

CPS cannot provide precise analysis for the program in Figure 5(a) for both a single input value of n, and for several values of n. The program state representing the unexpected behaviour of the program in Figure 5(a) is the value of x at line 7. CPS cannot precisely capture this behaviour because for some executions of the program, with input value 13, CPS will collect the state of the variable x after the execution of lines 2 and 6, but for the other execution CPS will collect the state of the variable x after the execution of lines 2 and 5. Furthermore, for the cases where CPS collects the state of the variable x after the execution of lines 2 and 5, the value of variable x will vary based on the value of r and. In order for the CPS analysis to become more precise samples of the different possible variable states from each possible dynamic program slice in the program are required. Next, we discuss how PSDFs achieve this functionality.

2.3 Program slice distribution functions (PSDFs)

Our approach involves application of Monte Carlo sampling to the different possible dynamic program slices for a stochastic simulation and the different possible values of variable states within each possible dynamic program slice given a fixed input (Berg, 2004).

Generating and applying PSDFs to enable precise CPS analysis. The method for generating PSDFs and applying them to CPS proceeds as follows:

1. The user identifies the unexpected behaviour in the stochastic simulation with the static program slicing criterion that captures the program state of the unexpected behaviour. The user also specifies the causal influence threshold for the slice.
2. Using the static slicing criterion the stochastic simulation is preprocessed, as it is in traditional CPS.
3. The user specifies the different inputs of interest for which she/he wishes to analyse the unexpected behaviour. The user also specifies, *n*, the number of times to execute the simulation for each input.
4. Each input of interest is combined with the static slicing criterion. This forms a dynamic slicing criterion for each input of interest.
5. For each dynamic slicing criterion the stochastic simulation is executed *n* times to generate a PSDF.
   (a) For each execution the dynamic program slice is grouped with the dynamic program slice it matches exactly. If no such dynamic program slice exists, a new group is formed. Two dynamic program slices A and B match exactly if and only if they execute the same statements in the same order. Variable state data that accompanies a dynamic program slice does not play a role in the matching process.
   (b) The variable states within the dynamic program slice are stored along with the program statements that caused the changes in variable state. The result of Steps 5(a) and (b) is a probability distribution of the dynamic program slices and variable states within the dynamic program slices given the slicing criterion.
6. The user specifies the number of times, *k*, to sample each PSDF.
7. Each PSDF is sampled *k* times. Each sample from a PSDF requires two steps.
   (a) First, a dynamic program slice group is chosen from a uniform random distribution over all the different dynamic program slice groups within a PSDF.
   (b) Within the chosen dynamic program slice group, a variable state data sample is picked from a uniform random distribution over all the variable state data samples for the identified dynamic program slice group. The variable state data sample is added to the CPS data set.
8. Once all PSDFs are sampled *k* times the CPS data set is formed, and traditional CPS analysis is performed on the data set.

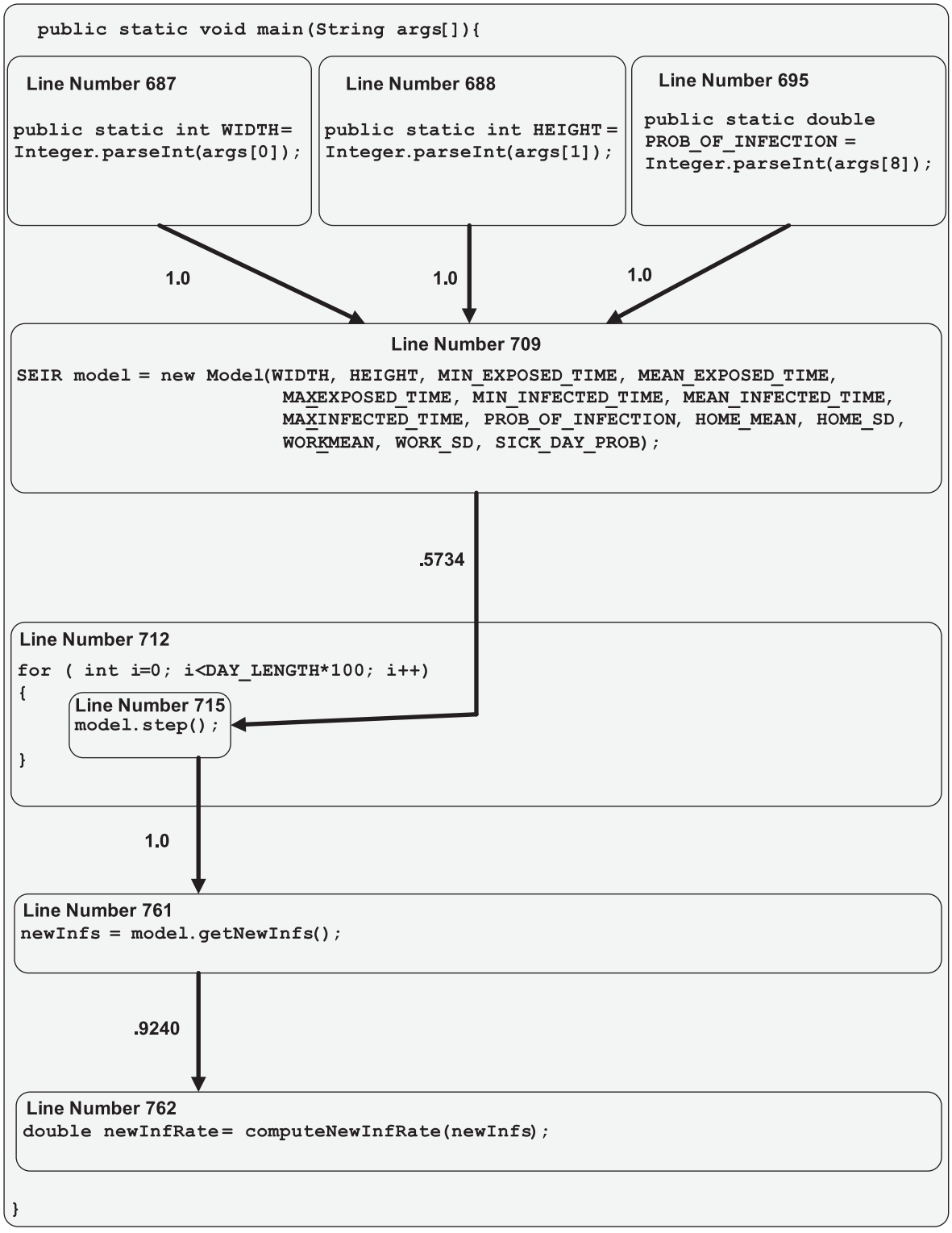

Figure 6 The causal program slice with threshold 0.5 with respect to line number 762, the computation of the rate of new infections in Dunham's simulation.

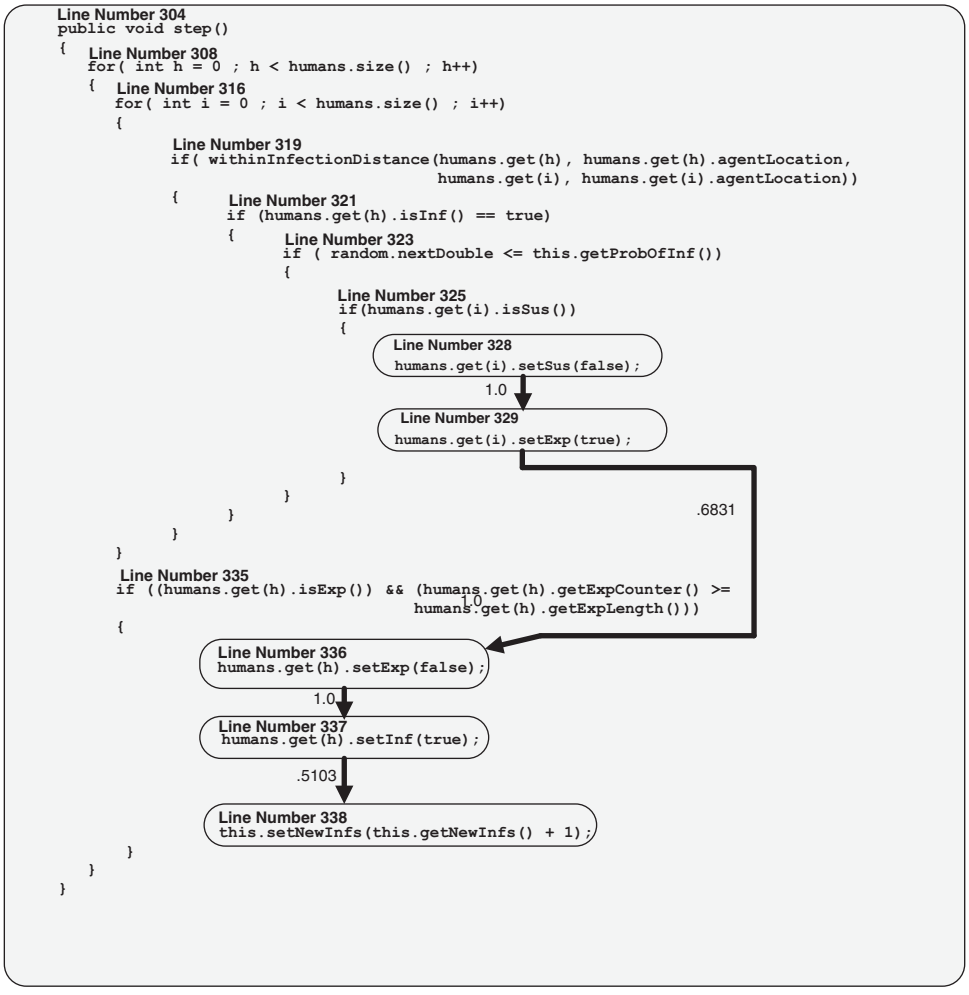

Figure 7 The causal program slice with a threshold of 0.5 for the function step with respect to line number 715.

This strategy balances the probability that a variable state affects the unexpected behaviour with the influence of the variable state, when it does affect the behaviour.

Applying precise CPS analysis to the Dunham simulation. To evaluate the more precise CPS analysis enabled by PSDFs we explored the previously described unexpected behaviour in the Dunham simulation shown in Figure 2(a) and (b). Recall, applying SAHT did not enable users to understand which input parameters influence the unexpected behaviour, how those input parameters cause state changes in variables in the simulation's source code, and how the state changes in those variables control the rate of new infections in Dunham's simulation.

CPS reveals which statements in the simulation source code, beginning with input parameters, have the strongest influence on the rate of new infection in the Dunham simulation. Line 762 captures the program state of interest for the unexpected behaviour. Figure 6 is the result of applying CPS with a causal influence threshold of 0.5. The function containing line number 762 comprises 91 lines of source code. Applying static program slicing to the program state representing the unexpected behaviour, the computation of new infection rate in line number 762, reduces the number of statements in the source code that influence the unexpected behaviour to 68. Applying precise CPS with a causal influence threshold of 0.5 to the computation of the new infection rate in line number 762 reduces the number of statements to seven!

Figure 6 also shows that the call to step (line number 715) in the conditional loop statement changes the simulation state. This call has a strong causal influence on the new infection rate in the simulation. The call to getNewInfs( ) uses the state change of model caused by step to change the state of newInfs. To better understand the influence of the variable states and program statements in the function step on the new infection rate, we apply CPS again to each function.

Figure 7 shows the state changes that occur in the step function that influence the rate of new infection in the simulation. The only program statements with a strong influence on the rate of new infection in the step function are

contained in several nested conditional control flow statements within a conditional loop statement. We discuss several of the conditional control flow statements further. Line 319 establishes whether an agent (humans.get(i)) is within the infection radius of the current agent (humans.get(h)). Here, the effect of the input parameters WIDTH and HEIGHT on the rate of new infection is revealed. Lower values for the input parameters WIDTH and HEIGHT create a smaller torus, which is more densely packed with agents. As a result more agents fall within the infection radius of infected agents. This causes the rate of new infections to rise in the simulation. Thus, the unexpected behaviour we observed in Figure 2 (a) and (b) is explained. The Dunham simulation predicts a more intense infectious period of SEIR disease spread because the 2-D torus becomes more densely packed with agents when the population is changed from 100 to 1000.

The CPS in Figure 7 also reveals the effect of the input parameter PROB_OF_INF on the rate of new infection. PROB_OF_INF specifies the probability an infected agent spreads the disease to an agent within the specified infection radius. A lower PROB_OF_INF results in fewer agents becoming immediately exposed. As a result fewer agents become infected in later time steps leading to a lower rate of new infection. Conversely, a higher PROB_OF_INF results in more agents becoming immediately exposed. Thus, more agents become infected leading to a higher rate of new infection.

The ability to identify the changes in variable state that have the strongest influence on the rate of new infection, and to associate those changes with program statements that cause them, allows us to understand how the width of the 2-D torus, height of the 2-D torus and probability of spreading infection parameters cause the source code to compute the rate of new infection in the agent-based simulation. PSDFs enabled CPS to perform this analysis more precisely than traditional CPS. The more precise CPS analysis sampled different dynamic program slices for each set of input parameters and different values of variable states within each dynamic program slice.

3. Conclusion

When an unexpected behaviour is first observed in an agent-based simulation, the prospect of explaining and then either validating or eradicating that behaviour can be daunting. Most users apply classic debugging techniques to identify program statements that cause agents to create the unexpected behaviour. Subsequently an explanation for the behaviour is formed, code is modified and the user iterates this process until satisfied. The process is manual and time-consuming. INSIGHT automates this process with a cohesive methodology built on SAHT, CPS and PSDFs.

In this process of hypothesizing, experimenting and drawing causal conclusions, SAHT addresses the hypothesizing step. By supporting iterative exploration of user defined conditions of interest, SAHT enables discovery of those conditions that have high correlation with an unexpected behaviour. Even with a viable set of hypotheses about origins of unexpected behaviours, the effort required to identify the causative relationships between conditions of interest and unexpected behaviours can be overwhelming without automated support. CPS and PSDF support transition from a viable set of hypotheses to quantified explanations for unexpected behaviours in agent-based simulations by revealing how pieces of the simulation source code causally influence the user identified conditions correlated with the unexpected behaviour.

INSIGHT has extended the state of the art for facilitating user understanding of unexpected agent-based simulation behaviour. While INSIGHT is not limited to agent-based simulations, it is particularly useful for them because of their frequent tendency to exhibit unexpected behaviours that cannot be directly attributed to specific blocks of code. INSIGHT can be used to provide insight for any simulation where the user is uncertain of the validity of a particular simulation behaviour. We see the broader applicability of INSIGHT as beneficial to the general simulation community because we expect broader interest to lead to higher quality tools for all.

References

Arthur W (1999). Complexity and the economy. *Science* **284**: 107–109.

Baciu A, Anason A, Stratton K and Strom B (2005). *The Smallpox Vaccination Program: Public Health in an Age of Terrorism*. Institute of Medicine of the National Academies: Washington, DC.

Balci O (1997). Principles of simulation model validation, verification, and testing. *Trans Soc Comput Simulat* **14**: 3–12.

Berg A (2004). *Markov Chain Monte Carlo Simulations and Their Statistical Analysis: With Web-based Fortran Code*. World Scientific Press: London.

Brogi A, Canal C and Pimentel E (2003). Behavioural types and component adaptation. In: Rattray C, Maharaj S and Shankland C (eds). *Proceedings of 10th International Conference on Algebraic Methodology and Software Technology*. Springer Berlin: Heidelberg, pp 42–56.

Carnahan J *et al* (2007). Using flexible points in a developing simulation of selective dissolution in alloys. In: Herderson S, Biller B, Hsieh M, Shortle J, Tew J and Barton R (eds). *Proceedings of the 2007 Winter Simulation Conference*. Institute of Electrical and Electronic Engineering: Piscataway, pp 891–899.

Cha A (2005). Computers simulate terrorism's extremes. In: Bennett P, Coleman M and Downie L (eds). *Washington Post*, 4 July 2005 Washington Post: Washington, D.C, p A1.

Chang F and Karamcheti V (2001). A framework for automatic adaptation of tunable distributed applications. *Cluster Comput* **4**(1): 49–62.

Cohen J (1988). *Statistical Power Analysis for the Behavioral Sciences*. L. E. Associates: Philadelphia.

Corning P (2002). The re-emergence of 'emergence': A venerable concept in search of a theory. *J Complexity* **7**(6): 18–30.

Diekmann O and Heersterbeek J (2000). *Mathematical Epidemiology of Infectious Diseases*. Wiley: New York City.

Dunham J (2005). An agent-based spatially explicit epidemiological model in MASON. *J Artif Soc Soc Simulat* **9**(1): 3.

Elderd B, Dukic V and Dwyer G (2006). Uncertainty in predictions of disease spread and public health responses to bioterrorism and emerging diseases. *Proc Natl Acad Sci* **103**(42): 15693–15697.

Eubank S *et al* (2004). Modeling disease outbreaks in realistic urban social networks. *Nature* **2541**(249): 180–184.

Garcia A (2000). Orthogonal sampling formulas: A unified approach. *SIAM Rev* **42**: 499–512.

Gore R and Reynolds P (2008). Applying causal inference to understand emergent behavior. In: Mason S, Hill R, Moench L and Rose O (eds). *Proceedings of the 2008 Winter Simulation Conference*. Institute of Electrical and Electronic Engineering: Piscataway, pp 712–721.

Gore R and Reynolds P (2009). Causal program slicing. In: Reynolds P (ed) *Parallel Distr Simulat 2009*. Society for Modeling and Simulation International: San Diego, pp 19–26.

Gore R, Reynolds P, Tang L and Brogan D (2007). Explanation exploration: Exploring emergent behavior. In: Perumalla K (ed) *Parallel Distr Simulat 2007*. Society for Modeling and Simulation International: San Diego, pp 113–122.

Gschwind T (2002). *Adaptation and composition techniques for component-based software engineering* PhD thesis, Technischen Universi̇at Wien.

Haack C, Howard B, Stoughton A and Wells J (2002). Fully automatic adaptation of software components based on semantic specifications. In: Kirchner H and Ringeissen C (eds). *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*. Springer Berlin: Heidelberg, pp 83–98.

Hooke W and Pielke R (2000). *Prediction: Science, Decision Making, and the Future of Nature*. Island Press: Washington, D.C.

Li M and Muldowney J (1995). Global stability for the SEIR model in epidemiology. *Math Biosci* **125**(2): 155–164.

Loh W (1996). On Latin hypercube sampling. *Ann Stat* **24**(5): 2058–2080.

Lucas T, Sanchez S, Brown L and Vinyard W (2002). Better designs for high-dimensional explorations of distillations. In: Horne G and Johnson S (eds). *Proceedings of Maneuver Warfare Science 2002*. United States Marine Corps Project Albert: Quantico.

Montgomery D (2004). *Design and Analysis of Experiments* 6th edn, Wiley & Sons: Indianapolis, IN.

National Science Foundation (2006). *Simulation-based engineering science: Revolutionizing engineering science through simulation* Report of the NSF Blue Ribbon Panel on Simulation-Based Engineering Science.

Pearl J (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press: Cambridge.

Reiher P, Guy R, Yarvis M and Rudenko A (2000). Automated planning for open architectures. In: Hutchinson D (ed) *Proceedings of the Third IEEE Conference on Open Architectures and Network Programming*. Institute of Electrical and Electronic Engineering: Piscataway, pp 17–20.

Reynolds P, Spiegel M, Liu X and Gore R (2007). Validating Evolving Simulations in COERCE. *Lect Notes Comput Sc* **4487**: 1238–1245.

Spirtes P, Glymour C and Scheines R (2001). *Causation, Prediction, and Search*. Springer Verlag: New York City.

Tip F (1995). A survey of program slicing techniques. *J Program Lang* **3**(3): 121–189.

Trenouth J (1991). A survey of exploratory software. *Comput J* **34**(2): 153–163.

Van der Walt C and Barnard E (2006). Data characteristics that determine classifier performance. In: Louw A, Kleynhans N and Zulu N (eds). *Proceedings of the 17th Annual Symposium of the Pattern Recognition Association of South Africa*. International Association for Pattern Recognition: Durham, pp 160–165.

Wielinga B (1978). AI programming methodology. In: Sleeman, D (ed). *Proceedings of the Artificial Intelligence and Simulation of Behavior Conference*. Society for the Study of Artificial Intelligence and Simulation of Behavior: Brighton, pp 355–374.

Weiser M (1984). Program slicing. *IEEE Trans Software Eng* **10**(4): 352–357.

Whipple C (1996). Can nuclear waste be stored safely at yucca mountain? *Sci Am* **274**(6): 72–79.

Zeigler B, Praehofer H and Kim T (2000). *Theory of Modeling and Simulation* 2nd edn, Academic Press: Burlington.

Received 9 March 2009;
accepted 10 May 2009 after two revisions