Proceedings of the 2010 Winter Simulation Conference
B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, eds.

OUT-OF-ORDER EXECUTION AND STRUCTURAL EQUIVALENCE OF SIMULATION MODELS

Tobin A. Bergen-Hill
Ernest H. Page

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7539, USA

ABSTRACT

This paper revisits a technique for determining structural equivalence between simulation models. Specifically brought under scrutiny are the restrictions for applying a rule that expands a compound event vertex when converting a simulation graph model (SGM) into an extended SGM. By checking for interdependencies of state variables within the vertex, one can ensure that the logical structure of the original model is preserved during expansion, allowing for "out-of-order" execution of events - thus permitting a greater class of models to be deemed structurally (and behaviorally) equivalent. An example is provided of establishing structural equivalence between two discrete event simulations derived from the same model, which benefits from the revised expansion rule.

1 INTRODUCTION

There are various reasons for establishing the equivalence of two simulation models. From a practical standpoint, it is a means of verifying the reimplementation of an existing model. From a theoretical standpoint, establishing a homomorphism from a simulation to an equivalence class of models demonstrates that the simulator will correctly execute those models (Zeigler, Praehofer and Kim 2000).

A large body of previous work exists that explores the means to demonstrate model equivalence via various approaches. Finite-state systems and their equivalences have been well established with theoretical backing by Milner (1984); as part of his Calculus of Communicating Systems (CCS), he demonstrates how to establish *operational equivalence* of process expressions, where two terms across all contexts have the same value. van Glabbeek and Vaandrager (1987) presented various notions of equivalence on Petri nets via process algebras, including *bisimulation equivalence* – an equivalence relation between two transition systems. Overstreet (1982) defined a formalism, the Condition Specification (CS), which is concise enough to facilitate automated diagnosis of the model representation; after demonstrating that any CS has an equivalent Turing Machine specification, he then concluded that the problem of determining whether any two models are functionally equivalent is undecidable. Yücesan and Schruben (1992) outline the means to test whether two simulation models are *structurally equivalent* (a stronger condition than behavioral equivalence) using simulation graph models, a formulation based on *event graphs* (Schruben 1983). Axtell et. al (1996) presented their "docking" process, the use of statistical methods to compare the output of two similar agent-based models in order to establish different levels of equivalence, up to and including *numerical identity* (where the two models produce numerically identical results).

This paper will primarily focus on the structural equivalence technique as applied to discrete event simulations. The CCS and Petri nets literature primarily applies to process and activity models, which focus on observable physical entities in a system rather than the events (Schruben 1983). Since it is possible to map stochastic Petri nets onto event graphs (Schruben and Yücesan 1994), a discussion centered on

manipulation of event graphs should be of interest to those communities as well. And while the docking process can make conclusions against a given set of inputs, structural analysis demonstrates behavioral equivalence across all possible inputs – as long as one has access to the models' implementations.

2 SIMULATION GRAPH MODELS

Before discussing structural and behavioral equivalence of models, a review of event graphs and simulation graph models is necessary. Schruben (1984) introduced the *event graph* formalization to represent the system structure of event-scheduled models. Each vertex of the directed graph corresponds to a system event – namely, when a state variable of the system is changed. The directed edges indicate the scheduling (or cancelling) of one event after another, as well as the logical and temporal relationships between the events. This is represented via edge conditions that govern when the next event is instigated, and a *delay time* variable. Figure 1 illustrates the event graph notation for a simple system. Event A initializes the state variable X to the value of 1. After a delay time t has elapsed and condition P holds true, the next event B is scheduled to occur, incrementing the state variable X by 1. The terms that appear in brackets under each vertex denote the state transition functions f_v associated with that vertex.

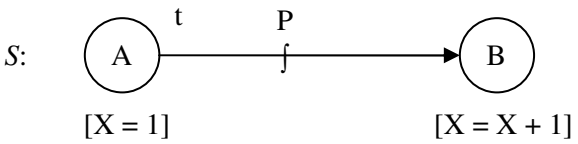

Figure 1: A simple example of an event graph.

When two events modify the same state variable, care must be taken to ensure that the events are not scheduled to execute simultaneously. So, an *event execution priority* is assigned to each vertex in the graph, where the higher priority event has precedence. [Note that model implementations are dependent on the execution environment, which may not provide constructs for explicitly stating event execution priorities. Such ordering will be determined based on the interdependency of the state changes, which will be addressed in section 3.]

Borrowing from graph theory, Yücesan and Schruben (1992) presented the *simulation graph*, a mathematically explicit definition of event graphs that has the following form:

$$G = (V(G), E_S(G), E_C(G), \Psi_G)$$

where $V(G)$ is the set of *event vertices* on graph G; $E_S(G)$ and $E_C(G)$ are the sets of *scheduling* and *cancelling edges*, respectively; and Ψ_G is the *incidence function* that associates each edge of G with an ordered pair of vertices. This led to the definition of a *simulation graph model* (SGM):

$$\mathcal{S} = (\mathcal{F}, \mathcal{C}, \mathcal{T}, \Gamma, G)$$

which is composed of the following indexed sets:

$\mathcal{F} = \{f_v : \text{STATES} \rightarrow \text{STATES} \mid v \in V(G)\}$, the set of *state transition functions* (or state changes) associated with vertex v,

$\mathcal{C} = \{C_e : \text{STATES} \rightarrow \{0, 1\} \mid e \in E_S(G) \cup E_C(G)\}$, the set of edge conditions,

$\mathcal{T} = \{t_e : \text{STATES} \rightarrow \mathcal{R}^+ \mid e \in E_S(G)\}$, the set of edge delay times,

$\Gamma = \{\gamma_e : \text{STATES} \rightarrow \mathcal{R}^+ \mid e \in E_S(G)\}$, the set of event execution priorities,

where STATES is defined in Zeigler, Praehofer and Kim (2000); and $\mathcal{R}^+$ denotes the set of nonnegative numbers.

2.1 Elementary Simulation Graph Models

A further distinction was made with regard to the composition of the edges and vertices of the SGM. An edge condition is categorized as *simple* if it consists of a single relation between two arithmetic expressions, or *compound* if it contains multiple relations joined by Boolean operators. Similarly, an event vertex is denoted as *simple* if it has at most one state variable change, or *compound* otherwise.

This extends to the categorization of the SGM as a whole. An *elementary simulation graph model* (ESGM), $\mathscr{S}^E$, is a simulation graph model, $\mathscr{S}$, that contains only simple event vertices and simple edge conditions. The process of converting any SGM into an ESGM is governed by four rules of *expansion* that preserves the logical structure of the original model. These expansion rules are in contrast to the event reduction rule in Schruben (1984), which eliminates "unnecessary" event vertices if they can be combined with a previous vertex.

While the first two rules cover the expansion of compound edge conditions, the third rule is of primary interest. It governs the expansion of a compound vertex; the possible outcomes are shown in Figure 2.

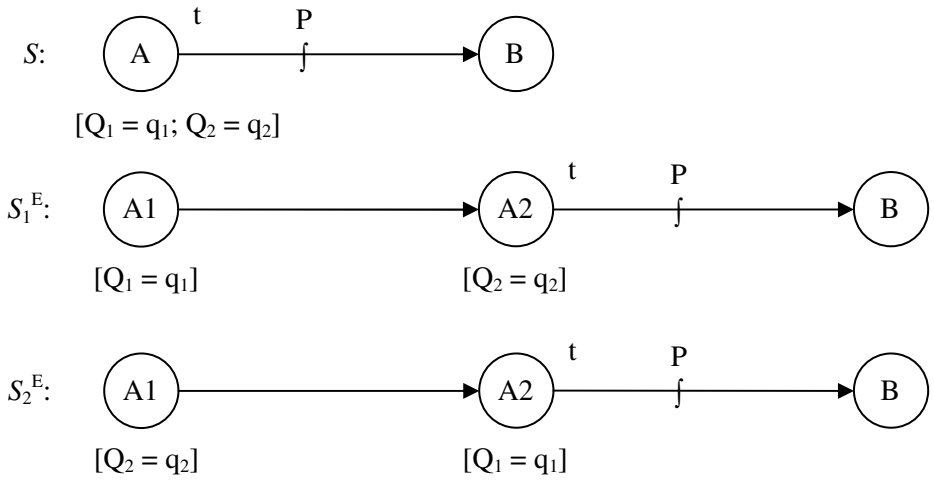

Figure 2: Rule 3 for expanding a compound vertex.

Presented here for the sake of discussion, the following is an excerpt from the caveat that appears in the paper after the illustration of rule 3:

> "Once again, note that the expansion of a compound event vertex is not unique. However, all expansions are isomorphic. In order to ensure correct execution of the event vertex, care should be exercised in expanding compound vertices. To this end, we can interpret the state transition function as a vector of state variable assignments. In computer implementations, this corresponds to a block of code where the flow of execution is sequential. Adhering to the original sequence of state variable assignments while expanding the vertex practically ensures that the logical structure of the original model is preserved." (Yücesan and Schruben 1992)

The assertion that "all expansions are isomorphic" is based on a proof of this rule that appears in Yücesan (1989). And yet, interdependencies between state transitions must be observed; otherwise the outcome will vary. For example, let Q1 and Q2 be the state variables X and Y. Suppose q1 is a constant value and q2 is the current value of X. The first expansion would result in X = 1 and Y = 1. But the second expansion would result in X = 1 and Y = X', the value of X prior to its assignment to 1. As stated in the excerpt above, "care should be exercised in expanding compound vertices."

To avoid this situation, the paper recommended "adhering to the original sequence of state variable assignments while expanding the vertex". However, such strict adherance may interfere with

demonstrating behavioral equivalence between models with slightly different orders of state variable assignments, the process for which is discussed in the next section.

2.2 Isomorphisms and Equivalence

After introducing the SGM formalism, Yücesan and Schruben (1992) moved on to equivalence. First, they deemed two SGMs $\mathscr{S}_1$ and $\mathscr{S}_2$ to be *isomorphic* if there exist the following mappings:

$\Theta : V(G_1) \rightarrow V(G_2)$ for the event vertices,
$\Phi_S : E_S(G_1) \rightarrow E_S(G_2)$ for the scheduling edges,
$\Phi_C : E_C(G_1) \rightarrow E_C(G_2)$ for the cancelling edges,
$\Lambda : \mathscr{F}_1 \rightarrow \mathscr{F}_2$ for the state transition functions,
$\Omega : \mathscr{C}_1 \rightarrow \mathscr{C}_2$ for the edge conditions,
$\chi : \mathscr{T}_1 \rightarrow \mathscr{T}_2$ for the delay times,
$\Delta : \Gamma_1 \rightarrow \Gamma_2$ for the event execution priorities.

The existence of such an isomorphism leads to *structural equivalence*:

Definition 1 Simulation graph models $\mathscr{S}_1$ and $\mathscr{S}_2$ are *structurally equivalent* if they have any elementary simulation graph models, $\mathscr{S}_1^E$ and $\mathscr{S}_2^E$, respectively, which are isomorphic.

They posit that such structural equivalence is a sufficient but not necessary condition for *behavioral equivalence*. Their definition of behavioral equivalence relies on a few more preliminary definitions. An *experimental frame* is "a specification of the conditions under which the system is observed or experimented with." (Zeigler, Praehofer and Kim 2000). These initial conditions determine the *execution* of the event vertices, each of which is dubbed an *occurrence* of the event vertex. The partially ordered set of the points in simulated time for these event occurrences is denoted as $T(\mathscr{E}, A)$ for experimental frame $\mathscr{E}$ and model A. An ordered set of states at each event occurrence is denoted as $S(\mathscr{E}, A)$.

Definition 2 Two simulation models A and B are *behaviorally equivalent* with respect to a subset of state variables if, within all experimental frames $\mathscr{E}$, $T(\mathscr{E}, A) \approx T(\mathscr{E}, B)$ and $S(\mathscr{E}, A) \approx S(\mathscr{E}, B)$.

Note that since these occurrences would include output events, the generation of the same output at the same time is a guarantee of numerical identity between the two models.

The two definitions are related in the following theorem (the proof of which appears in their paper):

Theorem If two SGMs are structurally equivalent then they must be behaviorally equivalent.

To summarize, the process of demonstrating structural (and behavioral) equivalence of two simulation models entails representing each as a SGM (with its associated event graph), expanding them into ESGMs, and determining an isomorphism between the two. If the two model implementations under investigation already have event graphs (i.e. the implementation was generated from the graph), then the process becomes much easier. Otherwise, since no tools currently exist for generating an event graph from any given implementation, the SGMs must be manually generated.

Problems arise when manually producing an event graph from a discrete event model implementation. Decomposing a block of code in a scheduled event into sub-event-graphs requires making arbitrary decisions of partitioning the code into event vertices. The ramifications of these decisions becomes clear when later attempting to establish an isomorphism between the ESGMs; the order of the event vertices affects the edges and the vertices' associated state transition functions, resulting in a failure to establish either an edge mapping Φ_S or a state transition function mapping Λ. This will become clearer in the next section.

2.3 Problematic Example: the Rebellion Models

To illustrate the issues in event graph generation from model implementations, sub-graphs from two implementations of Epstein's civil violence model (Epstein 2002) will be presented. One implementation uses the NetLogo multi-agent programmable environment (Wilensky 1999), and the other uses Repast-J (North, Collier, and Vos 2006). These sub-graphs are adapted from a more comprehensive treatment that can be found in an addendum to this paper (Bergen-Hill and Page 2010).

2.3.1 NetLogo Rebellion

The first implementation is the "Rebellion" model found in NetLogo's model library (Wilensky 2004). Figure 3 is a simulation graph G_{NR} depicting a subset of the SETUP event in the NetLogo implementation, which is effectively the first scheduled event in the simulation. The "events" in the graph all occur at simulation time 0. The state variables used in this example are defined as follows:

- patches is a two-dimensional array of NetLogo patch objects
- worldWidth is the number of patches across the X axis
- worldHeight is the number of patches across the Y axis
- seed is the random number generator seed
- FILE represents the NetLogo (.nlogo) file containing the model
- INPUT represents user input from the NetLogo user interface.

There are no edge conditions, and the edge delay times are all zero. No execution priorities have been set. The event descriptions are listed in Table 1. Note: the function RANDOM-SEED initializes the random number generator instance with the given seed.

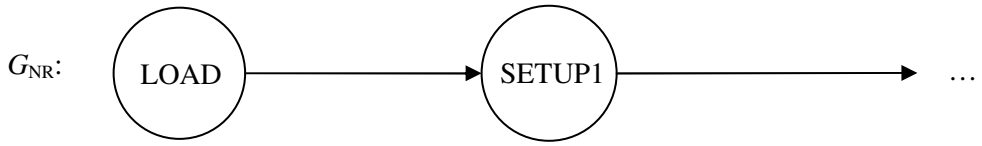

Figure 3: Event graph G_{NR} for the subset of the SETUP event from the NetLogo Rebellion model.

Table 1: Event table for the subset of the SETUP event from the NetLogo Rebellion model.

Event Type	Event Description	State Changes
LOAD	Read the values of worldWidth and worldHeight from the .nlogo file, then create the array of patches	worldWidth ← FILE worldHeight ← FILE patches ← create patch[worldWidth][worldHeight]
SETUP1	Get the value of seed from the user interface and initialize the MersenneTwisterFast with the seed	seed ← INPUT RANDOM-SEED(seed)

Expanding G_{NR} into an ESGM requires applications of expansion rule 3 for the event vertices LOAD and SETUP1. Strictly adhering to the sequence of instructions results in the particular ESGM shown in Figure 4.

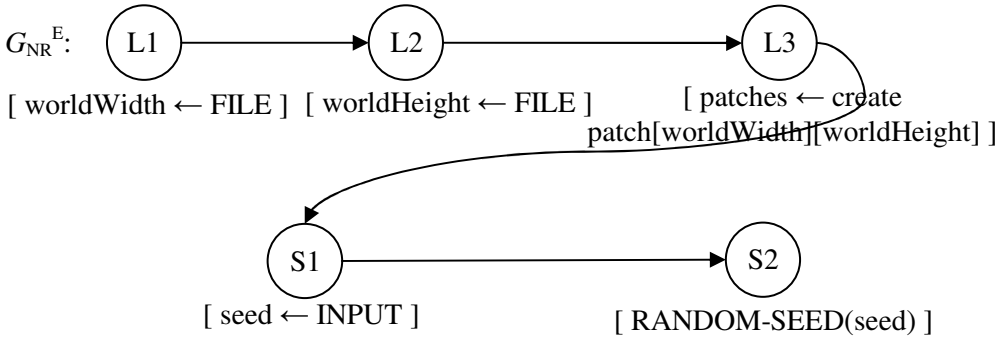

Figure 4: Elementary simulation graph G_{NR}^{E} for the SGM G_{NR}.

2.3.2 Repast-J Rebellion

Figure 5 is a simulation graph G_{RR} depicting a subset of the INIT event in a Repast-J implementation of the civil violence model, which was derived from the NetLogo implementation. Similar to the NetLogo model, INIT is the first event in the simulation, with all events occurring at simulation time 0. The state variables used in this example are defined as follows:

- patches is a two dimensional array of Repast-J Patch objects
- xSize is the number of patches across the X axis
- ySize is the number of patches across the Y axis
- rngSeed is the random number generator seed
- INPUT represents user input from the Repast-J user interface.

As with the NetLogo Rebellion graph G_{NR}, there are no edge conditions, and the edge delay times are zero. However, the execution priority of BEGIN is higher than DOMAIN-INIT, due to the dependency of events listed in Table 2.

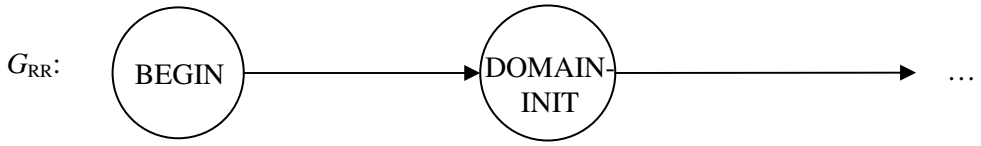

Figure 5: Event graph G_{RR} for the subset of the INIT event from the Repast-J Rebellion model.

Table 2: Event table for the subset of the INIT event from the Repast-J Rebellion model.

Event Type	Event Description	State Changes
BEGIN	Read the values of rngSeed, xSize and ySize from the user interface	rngSeed ← INPUT xSize ← INPUT ySize ← INPUT
DOMAIN-INIT	Initialize the MersenneTwisterFast with the seed and create the array of patches	RANDOM-SEED(rngSeed) patches ← create patch[xSize][ySize]

By adhering strictly to the order of instructions in the table above, an ESGM shown in Figure 6 is generated. Note how the event sequence differs from the NetLogo ESGM in Figure 4.

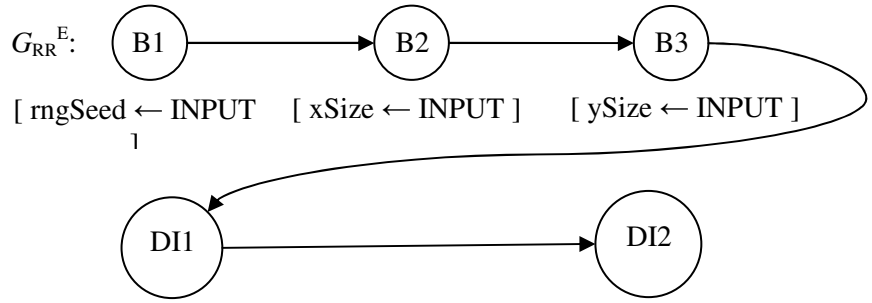

Figure 6: Elementary simulation graph G_{RR}^{E} for the SGM G_{RR}.

2.3.3 Attempting an Isomorphism

With the elemental simulation graph models in place, the next step is to determine the bijections for the isomorphism. The first problem arises when attempting to match the vertices between the ESGMs. Ignoring the content of the vertices' state transition functions, the two graphs are structurally identical, resulting in straightforward mappings for the vertices and scheduled edges:

$\Theta : V(G_{NR}^{E}) \rightarrow V(G_{RR}^{E})$

$\Theta(L1) = B1$
$\Theta(L2) = B2$
$\Theta(L3) = B3$
$\Theta(S1) = DI1$
$\Theta(S2) = DI2$

$\Phi_S : E_S(G_{NR}^{E}) \rightarrow E_S(G_{RR}^{E})$

$\Phi_S((L1,L2)) = (B1,B2)$
$\Phi_S((L2,L3)) = (B2,B3)$
$\Phi_S((L3,S1)) = (B3,DI1)$
$\Phi_S((S1,S2)) = (DI1,DI2)$

[Note that there are no canceling edges, so the mapping Φ_C is trivial.] But none of the vertices' corresponding state transition functions match between the graphs; the order is completely different.

Alternatively, if the focus is on the state transition functions, the mapping is straightforward:

$\Lambda : \mathscr{F}_{NR}^{E} \rightarrow \mathscr{F}_{RR}^{E}$

$\Lambda(f_{S1}) = f_{B1}\{$ rngSeed ← INPUT $\}$
$\Lambda(f_{L1}) = f_{B2}\{$ xSize ← INPUT $\}$
$\Lambda(f_{L2}) = f_{B3}\{$ ySize ← INPUT $\}$
$\Lambda(f_{L3}) = f_{DI2}\{$ patches ← create patch[xSize][ySize]$\}$
$\Lambda(f_{S2}) = f_{DI1}\{$ RANDOM-SEED(rngSeed) $\}$

But it then becomes unclear how to map the scheduling edges, since the vertices associated with the state transition functions occur in a different order. For example, in the following edge mapping, the initial edge in G_{NR}^{E} has a mapping to G_{RR}^{E}, but the other edges have no corresponding edges in G_{RR}^{E}, due to the difference in the state transition functions associated with each edge's vertices:

$\Phi_S : E_S(G_{NR}^{E}) \rightarrow E_S(G_{RR}^{E})$

$\Phi_S((L1,L2)) = (B2,B3)$
$\Phi_S((L2,L3)) = ?$ $\Phi_S((L3,S1)) = ?$ $\Phi_S((S1,S2)) = ?$

Regardless of the grouping of the implementation's code into event vertices, the resulting ESGMs will all run into the same conflict – as long as the events must retain the sequential order found in the implementation.

3 OUT-OF-ORDER EXECUTION

The traditional sequential execution of instruction by a processor was challenged in the 1960's by the introduction of *out-of-order completion/execution* techniques such as scoreboarding or the Tomasulo algorithm (Hennessey and Patterson 2007). Both techniques will resolve (or complete) *independent instructions* (i.e. ones that have no dependencies on earlier instructions) before dependent instructions. This results in the actual order of completed instructions differing from their original order (i.e. as it appeared in the program's source code).

In a simulation context, Page and Nance (1999) suggest an approach to implementing models represented in the Condition Specification (Overstreet, Page and Nance 1994) where the reordering of value-changing actions should be restricted to those that have no dependencies. The Condition Specification (CS) distinguishes three classes of state variables (which are referred to as "attributes" in the CS):

- Control attributes. Attributes that provide the information needed to determine when an action should occur. These are the attributes that occur in conditional expressions.
- Input attributes. Attributes that provide the data to be used to set new values for output attributes or to schedule future actions.
- Output attributes. Attributes that change value due to the action.

Building on these attribute classes, Page and Nance (1999) suggest the following definitions of conflicting model actions that in turn define what *dependency* means in this context:

Definition 3.1 *Two model actions a and b are said to be write/write conflicting if the intersection of their output attribute sets is non-empty.*

Definition 3.2 *Two model actions a and b are said to be read/write conflicting if the intersection of the input and control attribute sets of one and the output attribute set of the other is non-empty.*

Definition 3.3 *Two model actions a and b are said to be dependent if they are write/write or read/write conflicting. Otherwise a and b are independent.*

One useful tool for dependency analysis is the *action cluster incidence graph*, which is a directed graph in which the *action clusters* (i.e. groups of actions taken by the model whenever a condition is true) in the model are the nodes and the attributes that 'connect' them are the arcs (Nance and Overstreet 1987). These arcs represent the dependencies between actions; a lack of an arc indicates the actions are independent.

As an example, suppose a simple model had only two action clusters A and B, where A included the assignment to X (an input variable) and Y (a control variable). Y is used in a condition for moving on to B. If the model were represented as an event graph as part of a SGM S, Figure 7 shows the result. Since A would be a compound vertex, converting S to an ESGM would have two possible expansions (using the third rule of expansion).

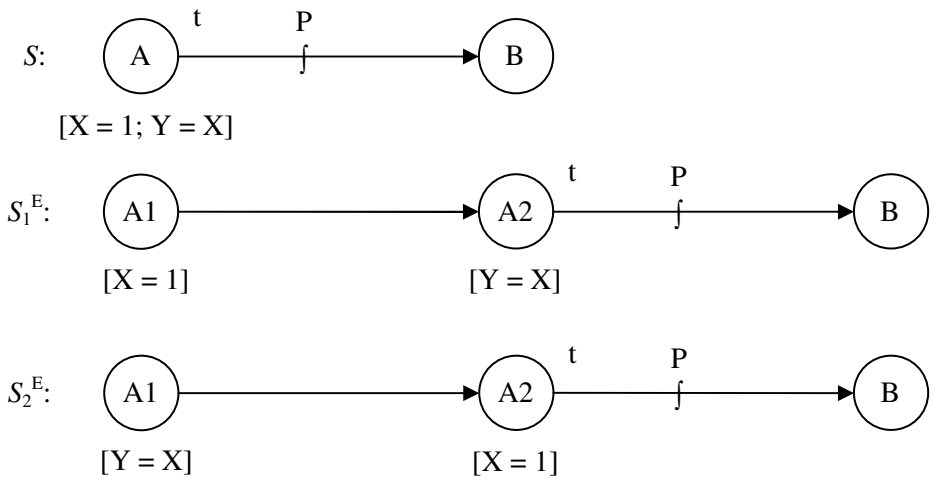

Figure 7: An event graph S for a simple model and two expansions of vertex A.

Using the example above, the corresponding action cluster incidence graph (ACIG) is depicted in Figure 8. Each action is in its own cluster because of the dependency. From inspection of the ACIG, the action {Y = X} is dependent on the action {X = 1} because of a read/write conflict, so the second expansion S_2^E is not permitted.

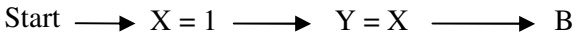

Figure 8: Action cluster independence graph for the event graph *S* in Figure 7.

By respecting these dependencies when reordering state variable changes, the expansion of compound event vertices can be performed while preserving the state of an event at a given simulation time – thus preserving structural (and behavioral) equivalence between the SGM and the ESGM. Also, by allowing more flexibility for the order of state variable changes, this permits a choice of possible expansions of the SGM into an ESGM that is more in line with another model's ESGM - thereby facilitating the generation of an isomorphism between the two models.

3.1 Revisiting Rule 3

With the advent of microprocessors such as the Pentium 4 that feature out-of-order execution (Hennessey and Patterson 2007), and given a methodology exists for identifying which instructions are dependent, it is recommended that the cautionary note following the explanation of expansion rule 3 be reworded to the following (with changes marked in *italics*):

> "Once again, note that the expansion of a compound event vertex is not unique. In order to ensure correct execution of the event vertex, care should be exercised in expanding com-pound vertices. To this end, we can interpret the state transition function as a vector of state variable assignments. In computer implementations, this corresponds to a block of code where the flow of execution *may occur in an out-of-order sequence*. *Varying the order of independent state variable assignments and* adhering to the original sequence of *dependent* state variable assignments while expanding the vertex practically ensures that the logical structure of the original model is preserved."

This will permit expansions that reorder simple vertices with independent state variable assignments while disallowing expansions that violate the dependencies.

Revisiting the summary of the process, demonstrating structural (and behavioral) equivalence of two simulation models now entails representing each as a SGM (with its associated event graph), *identifying dependencies (e.g. via construction of ACIGs)*, expanding the SGMs into ESGMs *while honoring the dependencies*, and determining an isomorphism between the two.

3.2 Return to Rebellion example

With the caveat to compound node expansion in place, it's time to return to the earlier attempt to determine structural equivalence of the two implementations of the Rebellion model.

Before expanding the NetLogo Rebellion model's SGM G_{NR} into an ESGM, an ACIG should be formed to guide the expansion choices for the two compound event vertices LOAD and SETUP1. Figure 9 depicts the dependencies of the action clusters in G_{NR}. Note that the initialization of the worldWidth, worldHeight and seed variables could occur out of the order listed in Table 1 without changing the model's behavior.

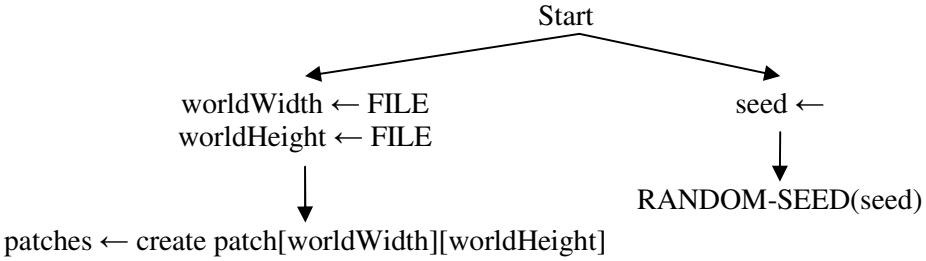

Figure 9: The ACIG for the actions listed in Table 1.

If the SETUP1 vertex were expanded so that the seed variable initialization was in its own vertex, the seed variable initialization could be combined with the LOAD vertex according to Event Reduction Rule 4(a) (Yücesan and Schruben 1992). Furthermore, the seed variable initialization could occur before the patches variable initialization – and even before the worldWidth and worldHeight variable initializations – due to its independence from these other actions. This permits the choice of the particular ESGM shown in Figure 10 out of the different possible expansions.

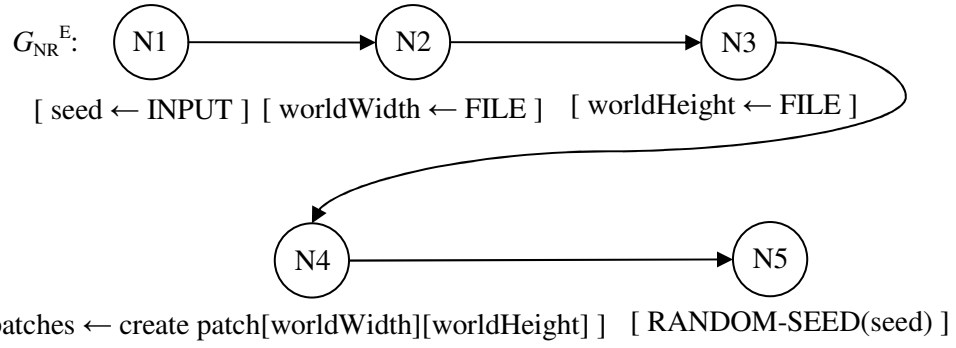

Figure 10: Elementary simulation graph G_{NR}^{E} for the SGM G_{NR}.

The left side of Figure 11 depicts the ACIG for the action clusters in the Repast-J Rebellion model's SGM G_{RR}. Analysis of the ACIG reveals that the initialization of the random number generator and the patch array creation actions are interchangeable because - although they depend on the variables initialized by input in the previous action cluster – they do not depend on each other. The interdependency of the patches and RNG initialization actions permits the choice of the ESGM depicted in the right side of Figure 11. Note its strong resemblance to the NetLogo ESGM G_{NR}^{E} - which is intentional.

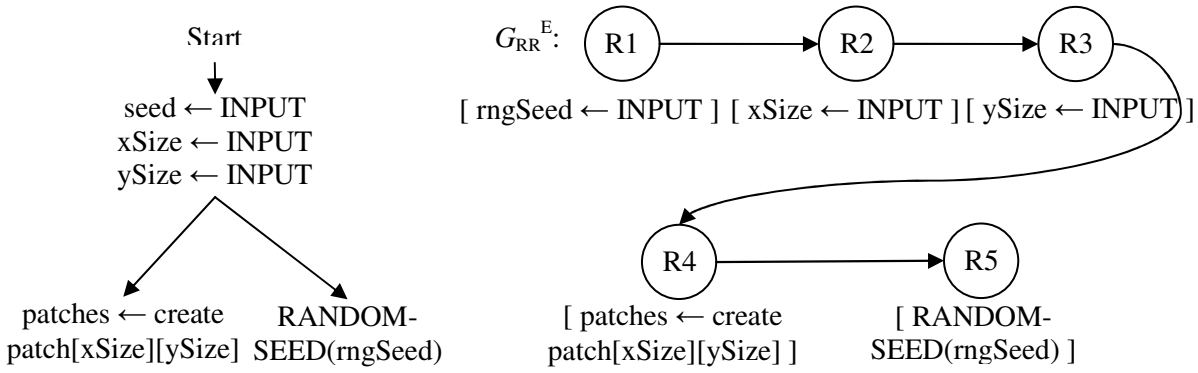

Figure 11: The ACIG and the elementary simulation graph G_{RR}^{E} for the SGM G_{RR}..

With the elemental simulation graph models in place, the final step is to determine the following bijections for the isomorphism between the ESGMs, as shown below:

$\Theta : V(G_{NR}^{E}) \rightarrow V(G_{RR}^{E})$

$\Theta(N1) = R1$
$\Theta(N2) = R2$
$\Theta(N3) = R3$
$\Theta(N4) = R4$
$\Theta(N5) = R5$

$\Phi_S : E_S(G_{NR}^{E}) \rightarrow E_S(G_{RR}^{E})$

$\Phi_S((N1,N2)) = (R1,R2)$
$\Phi_S((N2,N3)) = (R2,R3)$
$\Phi_S((N3,N4)) = (R3,R4)$
$\Phi_S((N4,N5)) = (R4,R5)$

$\Phi_C : E_C(G_{NR}^{E}) \rightarrow E_C(G_{RR}^{E})$ [Since there are no cancelling edges E_C for either graphs, this is trivial.]

$\Lambda : \mathscr{T}_{NR}^{E} \rightarrow \mathscr{T}_{RR}^{E}$

$\Lambda(f_{N1}) = f_{R1}\{$ rngSeed ← INPUT $\}$
$\Lambda(f_{N2}) = f_{R2}\{$ xSize ← INPUT $\}$
$\Lambda(f_{N3}) = f_{R3}\{$ ySize ← INPUT $\}$
$\Lambda(f_{N4}) = f_{R4}\{$ patches ← create patch[xSize][ySize]$\}$
$\Lambda(f_{N5}) = f_{R5}\{$ RANDOM-SEED(rngSeed) $\}$

$\Omega : \mathscr{C}_{NR}^{E} \rightarrow \mathscr{C}_{RR}^{E}$ [Due to the lack of edge conditions in the two models, this mapping is trivial.]

$\chi : \mathscr{T}_{NR}^{E} \rightarrow \mathscr{T}_{RR}^{E}$ [Since the edge delay time $\mathscr{T}$ is zero for all edges in both graphs, this is trivial.]

$\Delta : \Gamma_{NR}^{E} \rightarrow \Gamma_{RR}^{E}$ [The execution priority Γ is the same value for all edges in both graphs; so this is trivial.]

With the existence of the above isomorphism – and the revised restrictions on applying expansion rule 3 - the initial portion of the two Rebellion model implementations are structurally equivalent, and thus behaviorally equivalent.

4 CONCLUSION

The determination of structural equivalence between simulation models, as outlined by Yücesan and Schruben (1992), can benefit from the use of dependency analysis techniques - such as action cluster incident graphs. By permitting the reordering of independent state variable assignments during expansion

of compond vertices, a larger class of model implementations can be proven to be structurally (and behaviorally) equivalent.

ACKNOWLEDGEMENTS

Approved for Public Release: 10-2458. Distribution Unlimited. ©2010 The MITRE Corporation. All rights reserved. The authors would like to thank Enver Yücesan and Lee Schruben for their feedback on this paper and for their invaluable contributions to computer simulation.

REFERENCES

Axtell, R., R. Axelrod, J. Epstein and M. Cohen. 1996. Aligning Simulation Models: A Case Study and Results. *Computational and Mathematical Organization Theory* 1(2):123-141.

Bergen-Hill, T., and Page, E. 2010. Addendum. Avaiable via <http://www.bergen-hill.com/papers/BehEq_WSC10_Addendum.pdf> [accessed March 2010]

Hennessey, J. L. and D.A. Patterson. 2007. *Computer Architecture : a Quantitative Approach.* 4th ed. San Francisco, California: Morgan Kaufman Publishers.

Milner, R. 1984. A complete inference system for a class of regular behaviors. *Journal of Computer and System Sciences* 28(3):439-466.

Nance, R. and C.M. Overstreet. 1987. Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications. *Transactions of the Society for Comp. Simulation* 4(1):33-57.

North, M. J., N.T. Collier and J.R. Vos. 2006. Experiences Creating Thre Implementations of the Repast Agent Modeling Toolkit. *ACM Transactions on Modeling and Computer Simulation* 16(1):1-25.

Overstreet, C.M. 1982. Model Specification and Analysis for Discrete Event Simulation, PhD Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, VA, December.

Overstreet, C.M., E.H. Page and R.E. Nance. 1994. Model Diagnosis using the Condition Specication: From Conceptualization to Implementation. *Proceedings of the 1994 Winter Simulation Conference*, 566-573. Orlando, Florida.

Page, E. and R. Nance. 1999. Incorporating Support for Model Execution within the Condition Specification. *Transactions of the Society for Computer Simulation* 16(1):47-62.

Schruben, L. 1983. Simulation modeling with event graphs. *Comm. of the ACM* 26(11):957-963.

Schruben, L. and E. Yücesan.1994. Transforming Petri Nets into event graph models. *Proceedings of the 1994 Winter Simulation Conference*, 560-565. Orlando, Florida.

van Glabbeek, R.J. and F.W. Vaandrager. 1987. Petri net models for algebraic theories of concurrency. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proceedings of the PARLE conference*, Eindhoven, *Vol. II (Parallel Languages)*. Lecture Notes in Computer Science 259, pages 224-242. Springer-Verlag.

Wilensky, U. 1999. *NetLogo*. Evanston, Illinois: Center for Connected Learning and Computer-Based Modeling, Northwestern University.

Wilensky, U. 2004. *NetLogo Rebellion Model*. Available via <http://ccl.northwestern.edu/netlogo/models/Rebellion> [accessed March 2010]

Yücesan, E. 1989. Simulation graphs for the design and analysis of discrete event simulation models. Ph.D. thesis, Cornell University, Ithaca, New York.

Yücesan, E. and L. Schruben. 1992. Structural and Behavioral Equivalence of Simulation Models. *ACM Transactions on Modeling and Computer Simulation* 2(1):82-103.

Zeigler, B.P., H. Praehofer and T.G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex System Dynamics*. 2nd ed. San Diego, California: Academic Press.

AUTHOR BIOGRAPHIES

TOBIN A. BERGEN-HILL is a lead simulation modeling engineer for the MITRE Corporation. He received his Masters degree from University of Maryland in 1997. His research interests include agent based modeling and simulation, high performance computing and application of serious gaming technologies. His email address is <tbergenhill@mitre.org>.

ERNEST H. PAGE is a member of the technical staff of The MITRE Corporation. He received the Ph.D. in Computer Science from Virginia Tech in 1994. He serves on the editorial boards of SCS Simulation, SCS Journal of Defense Modeling and Simulation, and the Journal of Simulation. He has held the position of Secretary (1995-1997), Vice Chair (1997-1999) and Chair (1999-2001) of the ACM Special Interest Group on Simulation (SIGSIM), and has served as the ACM SIGSIM representative to the WSC Board of Directors since 2001. His email address is <epage@mitre.org>.