Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

THE NEED FOR USABLE FORMAL METHODS IN VERIFICATION AND VALIDATION

Ross Gore
Saikou Diallo

Virginia Modeling, Analysis & Simulation Center
Old Dominion University
Norfolk, VA 23529, USA

ABSTRACT

The process of developing, verifying and validating models and simulations should be straightforward. Unfortunately, following conventional development approaches can render a model design that appeared complete and robust into an incomplete, incoherent and invalid simulation during implementation. An alternative approach is for subject matter experts (SMEs) to employ formal methods to describe their models. However, formal methods are rarely used in practice due to their intimidating syntax and semantics rooted in mathematics. In this paper we argue for a new approach to verification and validation that leverages two techniques from computer science: (1) *model checking* and (2) *automated debugging*. The proposed vision offers an initial path to replace conventional simulation verification and validation methods with new automated analyses that eventually will be able to yield feedback to SMEs in a familiar language.

1 INTRODUCTION

The process of developing models and simulations is well established. A conceptual model is designed by a subject matter expert (SME) from careful consideration of a problem and its domain. Then, it is realized via a source code simulation through the implementation of interfaces, data structures and algorithms. Finally, the output of the simulation for a set of test cases is validated against historical data or other trusted sources (Pace 2000, Robinson 2004, Robinson 2006, Page, Canova, and Tufarolo 1997).

Unfortunately, naively following this approach has pitfalls (Sargent 2005, Birta and Özmizrak 1996). The design of a model that appeared complete and robust can become incoherent, incomplete and potentially invalid during simulation implementation. The exactness required by the tools which execute the simulation creates complexity which might lead to the design of the conceptual model being hidden in irrelevant implementation details. Exploring alternative models and alternative modeling questions becomes impossible because SMEs are unable to identify the respective modifications that need to be made to the simulation (Law 2009).

An alternative approach is to employ a precise and unambiguous notation for SMEs to describe their models. This approach, known as formal methods, has had a number of major successes but is rarely used in practice due to three obstacles. First, the notation of formal specification languages has a mathematical syntax that makes them unintuitive and potentially intimidating. Second, these tools demand more investment of effort than can be expected from most SMEs. Finally, the tools force attention to mathematical details that don't reflect to SMEs the fundamental properties of their modeling question at hand (Kurshan 1997, Harbola, Negi, and Harbola 2012, Gajski, Abdi, Gerstlauer, and Schirner 2009, Woodcock, Larsen, Bicarregui, and Fitzgerald 2009, Findler and Mazur 1990).

Currently there is a large amount of effort that goes into establishing the validity of simulation. A myriad of validation techniques can be reduced to how well simulation outputs match other trusted simulations, historical datasets or reality. While this approach is a reasonable starting point, it is incomplete. Ideally, the validity of a simulation would be established by ensuring that it is a consistent implementation of the SME's conceptual model. Furthermore, the SME's conceptual model would be validated to ensure that it is a sufficient representation of reality given the test cases and modeling question at hand.

An example helps elucidate the importance of validating a conceptual model. Consider, Ptolemy's geocentric model of celestial movement where Earth is the orbital center of all celestial bodies. The geocentric model served as the predominant cosmological model in ancient Greece because its predictions largely matched Grecian observations. It was useful but wrong. While, Copernicus' correct heliocentric model came a millennium later it had no more predictive capabilities in the eyes of the Grecians. This example illustrates the need for formal validation and verification in modeling and simulation - the ability to accurately predict is necessary but not sufficient for validity.

To ensure that a conceptual model is *valid* a formal proof of how the SME's modeling question requirements are satisfied within the model is needed. Furthermore, when the model demonstrates behaviors which are not expected, but do not violate the SME's requirements, automated methods to explore the conditions under which the behaviors are manifested are needed. The fields of formal methods and automated debugging within computer science provide technologies which can be extended and adapted to address these deficiencies. Formal methods attack the exactness needed for axiomatic validation head-on while automated debuggers provide users with an exploratory capability to gather insight into unexpected outcomes. In what follows we provide a background in each of these areas.

In the remainder of this paper we: (1) identify deficiencies in the current validation process, (2) propose an architecture to address the deficiencies and (3) present our initial results in realizing the architecture via two case studies. Ultimately, our approach envisions a toolset that provides the incrementality and immediacy of small-scale conceptual modeling with the depth and clarity of formal methods.

2 BACKGROUND

Formal methods and automated software debugging tools in computer science can be extended and applied to address the current deficiencies in the verification and validation process for models and simulations. Here, we review and provide a background in these areas.

2.1 Automated Debuggers

The problem of identifying faults is a challenge in computer science. While a number of approaches exist, we focus on statistical debuggers. Statistical debuggers require a set of test cases for a buggy program, corresponding execution traces, and a labeling of the execution traces as passing or failing. The buggy program must pass at least one test case and fail at least one test case. The execution traces typically reflect coverage of individual statements or they reflect the truth values of inserted predicates. The debuggers assign *suspiciousness* scores to these program elements to measure the likelihood that a given element contains a fault. Then, the program elements are ranked in descending order of suspiciousness and returned to the developer tasked with debugging. An example helps elucidate the inner workings and utility of statistical debuggers.

Figure 1 shows the program, `mid()`, and its test suite. The program `mid()` takes three integers as input and is required to output the median value. The function fails to properly identify the median number for some inputs because there is a fault in Statement 7. Statement 7 should read `m = x`, however it reads `m = y`.

Figure 1 illustrates the process of employing statistical debugging to localize this fault. The debugger begins by executing `mid()` for each of the test inputs shown at the top of Figure 1. The execution of `mid` for each test input is traced to record the statements that are executed. The columns below the test inputs reflect each execution trace: a black dot signifies that the statement was executed, the lack of a black dot signifies that the statement was not executed.

	Program Source Code	Test Inputs						Suspiciousness	Rank
	`mid() {`	3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 3		
	`int x, y, z, m;`	Execution Trace							
1	`read("Enter 3 numbers:", x, y, z);`	●	●	●	●	●	●	.16	7
2	`m = z;`	●	●	●	●	●	●	.16	7
3	`if (y < z)`	●	●	●	●	●	●	.16	7
4	`if (x < y)`	●	●			●	●	.25	3
5	`m = y;`		●					.00	13
6	`else if (x < z)`	●				●	●	.33	2
7	`m = y;  // THIS IS A BUG!`	●					●	*.50*	*1*
8	`else`			●	●			.00	13
9	`if (x > y)`			●	●			.00	13
10	`m = y;`			●				.00	13
11	`else if ( x > z)`				●			.00	13
12	`m = x;`							.00	13
13	`print("Middle number is: ", m);`	●	●	●	●	●	●	.16	7
	`}`	*Actual* vs. Specified							
		3 vs. 3	*2* vs. 2	*2* vs. 2	*5* vs. 5	*4* vs. 4	*1* vs. 2		
		Pass/Fail Label							
		P	P	P	P	P	F		

Figure 1: Statistical debugging example.

Once mid() is executed for a given test input, the actual output of the program is compared to the specified output. The actual and specified outputs for each test input are shown immediately below the execution trace. The actual output is written in *italics* while the specified output is underlined. These outputs determine if the corresponding execution trace is labeled as passing or failing. If the actual output matches the specified output then the execution trace passes, otherwise it fails. The result of applying this labeling process to each execution trace of mid() is shown in the bottom row of Figure 1.

Labeling each execution trace as passing or failing enables the suspiciousness and rank of each statement in the source code of mid() to be computed. Recall, suspiciousness measures how likely it is that a statement contains a fault. It is calculated by computing the ratio of the number of failing execution traces that include the statement to the number of total execution traces that include the statement. The suspiciousness of each statement is shown in second rightmost column of Figure 1.

The rightmost column of Figure 1 shows the rank of each statement. The ranking column reflects the order the statements in mid() would be returned based on suspiciousness. The rank is the maximum number of statements that would have to be examined. It is assumed that all statements with the same suspiciousness are examined together.

In Figure 1, Statement 7 is identified as the most suspicious statement (.50) because it is only included in two execution traces - one that passes and one that fails. Every other statement is included in at least two passing execution traces and no statement is included in more than one failing execution trace. As a result Statement 7 is returned first to a developer tasked with debugging mid(). It is expected that by isolating the statement, the developer will quickly recognize the fault and correct the program. This is the appeal of statistical debugging. It is capable of automatically limiting the number of statements developers must sort through in the debugging process.

2.2 Predicate-level Statistical Debuggers

In addition to profiling program statements, most statistical debuggers employ conditional propositions, or *predicates*, to record the values assigned to variables in an execution trace. The suspiciousness of these predicates is calculated in the same manner as the suspiciousness of Statement 7. The addition of predicates enables statistical debuggers to analyze the coverage of statements *and* values in execution traces. In theory and practice this has been shown to improve effectiveness (Liblit 2008, Liu, Yan, Fei, Han, and Midkiff 2005). We review the two different types of predicate schemes next (single variables vs. scalar pairs) and discuss the difference between elastic and static predicates.

2.2.1 Single Variable Predicates

The *single variable* scheme partitions the set of possible values that can be assigned to a variable x in a statement y. In most debuggers, three static predicates are employed to partition the values for each x_y: ($x_y > 0$), ($x_y = 0$) and ($x_y < 0$). These three predicates are static because the decision to compare the value of x_y to 0 in each of these predicates is made before the subject program has been executed for any test inputs. While the decision to compare all variable values to zero via these predicates may seem arbitrary it is effective in general-purpose software. Most general purpose applications are almost entirely composed of discrete numbers and boolean conditions. The values these data types take on are indicators where the sign distinguishes one outcome from another (Liblit 2008).

The three static predicates can also be complemented by the nine elastic predicates presented in Table 1. The elastic predicates use summary statistics of the values assigned to x_y during execution to create partitions that cluster together values which are a similar distance and direction from μ_{x_y}. Elastic predicates are designed to target faults in simulations because they: (1) expand or contract based on observed variable values and (2) do not employ a rigid notion of equality (Gore, Reynolds, and Kamensky 2011).

Table 1: Fundamental single variable elastic predicates.

$x_y > (\mu_{x_y} + 3\sigma_{x_y})$
$(\mu_{x_y} + 3\sigma_{x_y}) \geq x_y > (\mu_{x_y} + 2\sigma_{x_y})$
$(\mu_{x_y} + 2\sigma_{x_y}) \geq x_y > (\mu_{x_y} + \sigma_{x_y})$
$(\mu_{x_y} + \sigma_{x_y}) \geq x_y > \mu_{x_y}$
$\mu_{x_y} = x_y$
$(\mu_{x_y} - \sigma_{x_y}) \leq x_y < \mu_{x_y}$
$(\mu_{x_y} - 2\sigma_{x_y}) \leq x_y < (\mu_{x_y} - \sigma_{x_y})$
$(\mu_{x_y} - 3\sigma_{x_y}) \leq x_y < (\mu_{x_y} - 2\sigma_{x_y})$
$x_y < (\mu_{x_y} - 3\sigma_{x_y})$

2.2.2 Scalar Pairs Predicates

Multiple variables within a program can have important relationships that cannot be captured with a single variable scheme. Statistical debuggers capture the relationships among multiple variables by identifying invariants that are only violated when the subject program fails. The scheme that captures these invariants is the *scalar pairs* scheme (Liblit 2008).

In the static scalar pairs scheme, at each assignment to a variable x in a statement y, all other in-scope, same-typed local or global variables: $q_1, q_2, ..., q_i, ..., q_n$ are identified. For each pair of variables the static scalar pairs scheme compares the difference of a new value for x_y and the existing value of q_i to zero: (x - $q_i > 0$), (x_y - $q_i = 0$), (x_y - $q_i < 0$).

In contrast, the nine scalar pair elastic predicates presented in Table 2 use summary statistics of the difference between the new value of x_y and the existing value of q_i to create partitions that cluster together differences which are a similar distance and direction from $\mu_{x_y - q_i}$.

Table 2: Fundamental scalar pairs elastic predicates.

$x_y - q_i > (\mu_{x_y-q_i} + 3\sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + 3\sigma_{x_y-q_i}) \geq x_y - q_i > (\mu_{x_y-q_i} + 2\sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + 2\sigma_{x_y}) \geq x_y - q_i > (\mu_{x_y-q_i} + \sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + \sigma_{x_y-q_i}) \geq x_y - q_i > \mu_{x_y-q_i}$
$\mu_{x_y-q_i} = x_y - q_i$
$(\mu_{x_y-q_i} - \sigma_{x_y-q_i}) \leq x_y - q_i < \mu_{x_y-q_i}$
$(\mu_{x_y-q_i} - 2\sigma_{x_y-q_i}) \leq x_y - q_i < (\mu_{x_y-q_i} - \sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} - 3\sigma_{x_y-q_i}) \leq x_y - q_i < (\mu_{x_y-q_i} - 2\sigma_{x_y-q_i})$
$x_y - q_i < (\mu_{x_y-q_i} - 3\sigma_{x_y-q_i})$

When used in combination the static and elastic predicates within the scalar pairs and single variable schemes can improve the efficiency of the debugging process for software developers significantly. In Section 3 we will explore how the principles underlying these techniques can be abstracted to provide an effective simulation verification strategy for SMEs.

2.3 Formal Methods

Employing formal methods in the field of modeling and simulation validation is now new. In safety-critical software, where billions of dollars and millions of lives depend on correctness, formal methods are routinely applied (Bowen and Stavridou 1993). This requires specifying a model in a formal language and using the properties of that language to ensure that every statement of the model is either part of the language or it can be produced by the language (Tolk, Diallo, Padilla, and Herencia-Zapana 2013). The term *formal methods* encompasses a family of approaches that sometimes refer to the tools, processes or languages used in support of validation (Tofts and Birtwistle 1998, Aldini, Bernardo, Gorrieri, and Roccetti 2001). In this section we review two formal techniques: (1) specification languages and (2) model checking. In Section 3 we describe how these techniques can be combined via a universal interface to enable SMEs to employ them for simulation validation.

2.3.1 Formal Specification Languages

Formal specification is the process of describing a system and its desired properties using a language with a mathematically defined syntax and semantics. Some formal specification languages such as Z (Spivey 1988), VDM (Jones 1986), and Larch (Guttag, Horning, Garl, Jones, Modet, and Wing 1993) focus on specifying the behavior of sequential systems, where states are described in mathematical structures such as sets, relations, and functions. Other methods such as CSP (Hoare 1978), CCS (Milner 1982), Statecharts (Harel 1987), Temporal Logic (Lamport 1994), and I/O automata (Lynch and Tuttle 1987) focus on specifying system behaviors in terms of sequences, trees, or partial orders of events. More recently languages based on first-order logic have been developed. These languages treat relations between system entities as first class citizens and use relational composition as a powerful operator to combine entities. The essential constructs of these languages are (Jackson 2006):

- *Signatures* - reflect a collection of relations (called fields) and a set of constraints on their values. A signature may inherit fiends and constraints from other signatures.
- *Predicates* - capture behavior constraints through general formulas.
- *Facts* - formulas which taken no arguments and impose global constraints on the relations and the objects.
- *Assertions* - specify intended properties within a model.

2.3.2 Model Checking

Model checking can be seen as solving a constraint problem. Given a model composed of signatures, predicates and facts and one or more specified assertions it performs two types of analyses: (1) *model instantiation* and (2) *counterexample generation* (Jackson 2006).

1. *Model instantiation* attempts to generate a version of a model specified in a formal specification language that satisfies all the predicates and facts given by the user. In general, model instantiation helps catch errors where the user has over specified the model, by reporting, contrary to the user intent, when no model instance satisfying all of the specified predicates and facts exists.
2. *Counter example generation* catches errors of under-constraint, by showing model instances that are acceptable given the specified scope, structure, operation and preconditions of the model but which violate a user specified assertion.

We envision model instantiation and counter example generation working together through an accessible interface to enable an incremental validation process. The user starts with a minimal conceptual model, and performs a variety of instantiations to detect over-constraint. Intended consequences are formulated, with counterexamples suggesting additional constraints to be added to the specification. This process, especially visualizing it through an accessible interface, will help produce a conceptual model that only reflects the components required to answer the SME's modeling question.

3 OUR VISION

Our vision for addressing the deficiencies identified in this paper has two parts: (1) employing a universal language as an interface to formal methods for SME conceptual model validation and (2) abstracting the principles underlying statistical debuggers to enable SMEs to efficiently verify unexpected outcomes. Here we describe both components of this vision. We expect that these descriptions along with our case studies in sections 4 and 5 will enable future work realizing our vision for verification and validation.

3.1 Extending Formal Methods To SMEs Through a Universal Language

The most effective interfaces combine graphical user interfaces (GUIs) with natural language (NL) processing capabilities (Cohen 1992). This combination creates a universal language which achieves a separation of concerns for users. We propose to employ such a language to allow users to distinguish the semantics of the conceptual modeling problem at hand from the semantics of formal specification. This will entail less of a learning curve for SMEs than working with formal specification languages directly. Furthermore, studies show that the communication between domain-experts improves when universal languages are employed in practice (North, Collier, and Vos 2006, Visser 2008). Next, we describe how a universal language can provide an interface to formal methods in our envisioned architecture.

Figure 2 demonstrates our vision for simulation validation. SMEs would specify the structure, operations and requirements of their conceptual model in a universal language combining a speech and sketch interface. Once specified, the conceptual model would be sent to a translator to convert the SME's model specification to a formal model specification. A model checker would employ over and under constraint analysis to check if the SME's conceptual model is consistent with the requirements from the SME's modeling question. The results of the analysis would be translated back to the universal language and visualized for the user. By iteratively refining their conceptual model in this manner the SME is guaranteed to generate a valid conceptual model. Ultimately, this architecture enables a type of model development that combines the incrementality and immediacy of small-scale implementation with the depth and clarity of traditional formal methods. Its realization is needed in the M&S community. Next, we turn our focus to simulation verification.

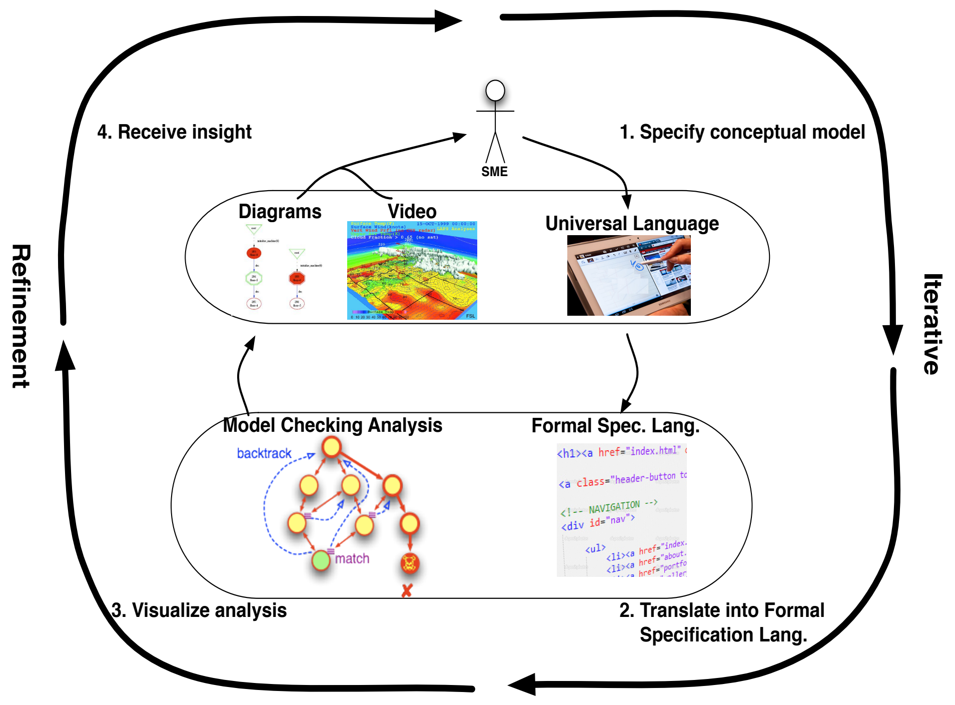

Figure 2: Architecture enabling SMEs to employ formal methods for verification.

3.2 Abstracting Statistical Debugging To Efficiently Explore Unexpected Outcomes

In the process of simulation verification SMEs need to understand and explain unexpected simulation outcomes to determine if the behaviors reflect an error or if they represent new knowledge. Ideally, a SME could identify parameters, outputs and *variables within the simulation* related to the outcome and automatically be given a ranked list of those conditions most relevant to the unexpected outcome. The ranked list would focus the SME's attention and greatly assist in understanding and explaining the behavior (Davis 2005).

Sensitivity analysis offers this capability to SMEs for simulation inputs and outputs. However, it cannot be applied to the variables within a model or simulation's internal structure. Similarly, statistical debugging is applied to all variables in the source code implementation of a simulation but this is frequently at too low a level to give most SMEs effective insight. We propose a tool that takes a log file created by a SME of variables which are hypothesized to be relevant to the unexpected outcome at different time steps or over different simulation runs. Along with the values of the SME identified variables each entry in the log file also must indicate if the unexpected outcome is manifested. Then the tool would perform static and elastic single variable and scalar pairs analysis to isolate those conditions most correlated with the presence of the unexpected outcome. By abstracting statistical debugging to this level its analysis is focused only on SME conditions of interest and it is applicable to all simulation platforms and all modeling paradigms.

4 MODEL CHECKING CASE STUDY: COALITION BATTLE MANAGEMENT LANGUAGE

Coalition Battle Management Language (C-BML) is defined as the unambiguous language used to: (1) command and control forces and equipment conducting military operations and (2) provide for situational awareness and a shared, common operational picture. It is domain-specific and employs custom XML tags to enable domains users to naturally define rigorous, well-documented, situational contexts, mission tasks and entity reports which reflect a subset of the Command & Control Information Exchange Data Model

(C2IEDM) (Tolk, Diallo, and Turnitsa 2007). There is a need to check C-BML documents to determine if tasks specified in missions are contradictory of military doctrine or can lead to failure (Tolk, Diallo, and Turnitsa 2006). However, this need has not been addressed. In what follows we present a solution to this problem using formal methods. Our solution exploits the domain-specific nature of XML. Users can naturally express Command & Control orders in C-BML and ultimately are provided with visual feedback about the logical consistency of their specified orders.

One C-BML plan describes the steps required by a Multi-National Company to take possession of Oscar 1, a house surrounded by an enemy force. The steps required to complete the mission are:

1. Move to a secure area and place a device monitoring Oscar 1.
2. Move along a discrete path to a location close to Oscar 1 to settle in for an assault.
3. Assault the enemies at Oscar 1 and seize possession of the house.
4. Fall north of Oscar 1 and install a monitoring device to the North, West and East of the seized location.

If steps 1-4 are successfully completed in order the invasion is complete and the Multi-National Company mission is considered successful. If the enemy ENI forces overtake the company, then the mission is considered a failure. The model is used by military organizations to explore various rules of engagement and resulting casualty scenarios for the Multi-National Company.

A C-BML document specifies each of the entities included in the scenario: the constituents of the Multi-National Company (US and French Platoon), Oscar 1, phase line, the enemy forces and the environment (forests, hills, etc). Then, the location and associations between these entities are flushed out via reports in C-BML. Finally, each step within the mission is expressed in C-BML tasks. We employ an XML parser to partition a C-BML document into its three components: *contexts*, *tasks* and *reports*. Once parsed from C-BML each context, task and report is converted into a formal specification into the formal specification language Alloy (Jackson 2006). This conversion is performed through the application of a series of algorithm shown in (Gore and Diallo 2013).

Using our algorithms the C-BML scenario can be translated into an Alloy model and intricacies within the mission can be explored. In this case study we explore how *rules of engagement* (ROEs) affect the success of the MultiNational Company. *Rules of engagement* are the directives issued by competent military authority that delineate the circumstances and limitations under which military forces will initiate and/or continue combat engagement with other forces encountered. These directives are intended to reduce the chance of friendly fire incidents and recognize international law regarding the conduct of war, particularly the need to protect civilians.

However, without validation these restrictions can limit the ability of commanders and companies to accomplish mission plans. Using our approach to usable formal methods, SMEs can explore if all the tasks within the mission can be carried out if the Multi-National Company never uses force against the Enemy ENI force. Figure 3 reflects output from the Alloy Analyzer model checker, visualized by a C-BML support tool, which reveals that all the tasks cannot be carried out if the Multi-National Company does not use force. The Multi-National Company (shown in blue) advances toward the Enemy ENI force but cannot assault the enemy and seize possession of the house. The company reaches an impasse where they surround the enemy but cannot progress further.

These types of subtleties in mission planning are notoriously difficult to explore at the scenario specification level but incredibly important to achieve success (Spiegel, Reynolds Jr, and Brogan 2005). However, our approach to making formal methods more usable enables: (1) different rules of engagement to be tested at the specification level and (2) users to determine exactly if and how the rules of engagement inhibit or guarantee mission success. This is a crucial step in enabling SMEs in military domains, regardless of their familiarity with formal methods, to produce valid conceptual models. In the next case study we explore our vision of abstracted statistical debugging to derive insight into an agent-based model of obesity exhibiting unexpected behavior.

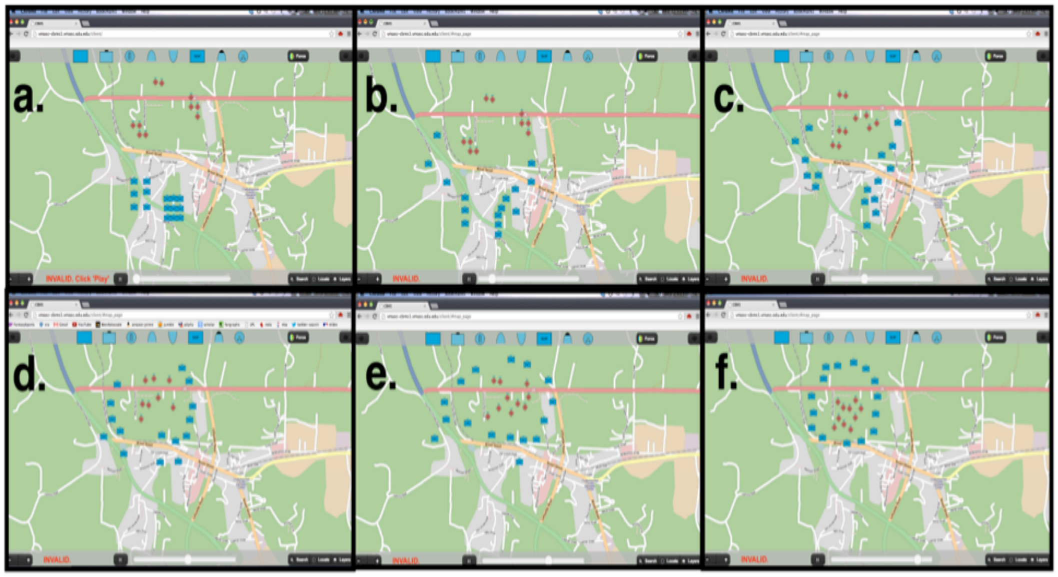

Figure 3: A counterexample generated by the Alloy Analyzer, visualized by a C-BML support tool.

5 STATISTICAL DEBUGGING CASE STUDY: AGENT-BASED OBESITY MODELING

In this case study we employ an agent-based model which is being developed to study how the availability of different restaurant choices in an area effect obesity (Tolk, Diallo, Padilla, and Herencia-Zapana 2013). Our obesity model contains four agent populations: people, homes, restaurants, and workplaces. Model inputs include the number of people, the eating habits of the people, the starting age and weight levels of the people, the number of workplaces, the number of restaurants, and the types of restaurants. Everyday each person eats three meals, which are obtained from the restaurants. Each meal contributes a number of calories to the individual based on the restaurants type. The restaurants that the people can eat at depend on their current location, either at home or at work, and the restaurants that are around their current location. At the end of every week, each persons weight is adjusted based on the number of calories that they consumed during the week compared against the number of calories that they needed to maintain their weight level. Calorie levels are reset each week and the obesity ranking of each individual is tracked over time.

The model reproduces historical data sets for different cities in the United States. Each city specifies the restaurant choices that are available and the data sets measure the percentages of the city population which are obese, overweight, normal weight and underweight. However, in some of the cities individual agents reach weights over 1,000 lbs. This occurs at a significantly higher rate than it does in the population of the United States. Here we apply abstracted statistical debugging to identify those conditions which are most correlated with the generation agents weighing more than 1,000 lbs or *super-gainers*.

The simulation developer identified seven agent-level conditions composed of parameters, internal simulation variables and model outputs that could influence super-gainers. These conditions are:

1. *Age* - The age of the agent.
2. *Height* - The height of the agent.
3. *Weight* - The weight of the agent.
4. *BMR* - The Basal Metabolic Rate (BMR) for the agent. BMR is the number of Calories the agent's body uses when in a resting state for a complete day.
5. *BMI* - The Body Mass Index (BMI) for the agent. BMI is used as a screening tool to identify possible weight problems.

6. *Calories* - The average number of calories the agent consumed in a week.
7. $LifeExp$ - The life expectancy of the agent given their height, weight and eating habits.

We generate elastic and static single variable and scalar pairs predicates from the seven conditions and track the predicates for each agent over a model run for one of the cities with a large number of super-gainers. If the agent is a super-gainer they are treated as a failing test case. If the agent is not a super-gainer they are treated as a passing test case.

The most suspicious predicate from uncovered by our abstracted statistical debugging analysis is the static scalar pairs predicate: $Age > LifeExp$. It has a suspiciousness of 1.00. Recall, from the discussion on suspiciousness in Section 2 this means that every agent who has an age greater than their life expectancy is a super-gainer. This feedback was relayed to the simulation developer who did not realize such a condition could occur in the model. Given the variables involved within the condition he was quickly able to locate and correct the simulation's implementation. In resulting runs there were significantly fewer super-gainers generated by the model.

The cause of super-gainers in the model seems clear enough once found. If obese agents are capable of outliving their life expectancy they will continue to gain weight in perpetuity. However it has been present and undiscovered for months in the model. Many bugs are obvious only once one knows where to look. The adapted statistical debugging results directed us to one condition (e.g., $Age > LifeExp$) out of the thousands of lines in the model as a whole.

Although we are still learning about the capabilities of this system and how to interpret its results, we believe that abstracting statistical debugging will make the process of finding and fixing the causes of unexpected outcomes in simulations more efficient and more responsive to the needs of SMEs.

6 CONCLUSION

The process of developing, verifying and validating models and simulations should be straightforward. Unfortunately, following conventional development approaches can render a model design that appeared complete and robust into an incomplete, incoherent and invalid simulation during implementation. An alternative approach is to employ formal methods for SMEs to describe their models. However, this approach is rarely used in practice due to the intimidating syntax and unfamiliar semantics of model checkers and theorem-provers.

As a result we have identified the need to extend and adapt formal verification and validation approaches to the modeling and simulation audience. A universal language combining speech and sketch input can be employed to provide an interface to these tools. Furthermore, the methodology behind statistical debuggers can be abstracted and applied to simulation log files to efficiently explore unexpected outcomes that do not violate the requirements specified by the SMEs modeling questions. Our case studies provide a blue print of how to move forward in these endeavors and in the future we expect that efforts within the modeling and simulation community will continue to realize this vision.

ACKNOWLEDGMENTS

We are grateful for help from our colleagues at The Virginia Modeling, Analysis and Simulation Center (VMASC) in shaping the ideas that are presented in this paper.

Ross Gore
Old Dominion University
Email: rgore@odu.edu

Saikou Diallo
Old Dominion University
Email: sdiallo@odu.edu

REFERENCES

Aldini, A., M. Bernardo, R. Gorrieri, and M. Roccetti. 2001. “Comparing the QoS of Internet audio mechanisms via formal methods”. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 11 (1): 1–42.

Birta, L., and F. Özmizrak. 1996. “A knowledge-based approach for the validation of simulation models: the foundation”. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 6 (1): 76–98.

Bowen, J., and V. Stavridou. 1993. “Safety-critical systems, formal methods and standards”. *Software Engineering Journal* 8 (4): 189–209.

Cohen, P. R. 1992. “The role of natural language in a multimodal interface”. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*, 143–149. ACM.

Davis, P. K. 2005. “New paradigms and new challenges”. In *Proceedings of the 37th conference on Winter simulation*, WSC ’05, 1067–1076: Winter Simulation Conference.

Findler, N., and N. Mazur. 1990. “A system for automatic model verification and validation”. *Transactions of the Society for Computer Simulation International* 6 (3): 153–172.

Gajski, D. D., S. Abdi, A. Gerstlauer, and G. Schirner. 2009. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated.

Gore, R. and Diallo, S. 2013. “CBML-To-Alloy Translation Algorithms”. http://www.cs.virginia.edu/~rjg7v/CBML-To-Alloy-Algorithms.pdf. [Online; accessed 19-June-2013].

Gore, R., P. F. Reynolds, Jr., and D. Kamensky. 2011. “Statistical Debugging with Elastic predicates”. In *Proceedings of the 26th IEEE/ACM international Conference on Automated software engineering*, ASE ’11, 273–282. New York, NY, USA: ACM.

Guttag, J. V., J. J. Horning, W. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. 1993. “Larch: languages and tools for formal specification”. In *Texts and Monographs in Computer Science*. Citeseer.

Harbola, A., D. Negi, and D. Harbola. 2012. “Infinite Automata And Formal Verification”. *International Journal* 2 (3).

Harel, D. 1987. “Statecharts: A visual formalism for complex systems”. *Science of computer programming* 8 (3): 231–274.

Hoare, C. A. R. 1978. “Communicating sequential processes”. *Communications of the ACM* 21 (8): 666–677.

Jackson, D. 2006. *Software Abstractions: logic, language, and analysis*. MIT press.

Jones, C. B. 1986. *Systematic software development using VDM*, Volume 66. Prentice Hall International.

Kurshan, R. P. 1997. “Formal verification in a commercial setting”. In *Proceedings of the 34th annual Design Automation Conference*, 258–262. ACM.

Lamport, L. 1994. “The temporal logic of actions”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16 (3): 872–923.

Law, A. M. 2009. “How to build valid and credible simulation models”. In *Simulation Conference (WSC), Proceedings of the 2009 Winter*, 24–33. IEEE.

Liblit, B. 2008. “Cooperative debugging with five hundred million test cases”. In *Proceedings of the 2008 International Symposium on Software testing and analysis*, ISSTA ’08, 109–120. New York, NY, USA: ACM.

Liu, C., X. Yan, L. Fei, J. Han, and S. P. Midkiff. 2005, September. “SOBER: statistical model-based bug localization”. *SIGSOFT Softw. Eng. Notes* 30:286–295.

Lynch, N. A., and M. R. Tuttle. 1987. “Hierarchical correctness proofs for distributed algorithms”. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 137–151. ACM.

Milner, R. 1982. *A calculus of communicating systems*. Springer-Verlag New York, Inc.

North, M. J., N. T. Collier, and J. R. Vos. 2006. "Experiences creating three implementations of the repast agent modeling toolkit". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 16 (1): 1–25.

Pace, D. K. 2000. "Ideas about simulation conceptual model development". *Johns Hopkins APL technical digest* 21 (3): 327–336.

Page, E. H., B. S. Canova, and J. A. Tufarolo. 1997. "A case study of verification, validation, and accreditation for advanced distributed simulation". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 7 (3): 393–424.

Robinson, S. 2004. *Simulation: the practice of model development and use*. Wiley.

Robinson, S. 2006. "Conceptual modeling for simulation: issues and research requirements". In *Proceedings of the 38th conference on Winter simulation*, 792–800. Winter Simulation Conference.

Sargent, R. G. 2005. "Verification and validation of simulation models". In *Proceedings of the 37th conference on Winter simulation*, 130–143. Winter Simulation Conference.

Spiegel, M., P. F. Reynolds Jr, and D. C. Brogan. 2005. "A case study of model context for simulation composability and reusability". In *Simulation Conference, 2005 Proceedings of the Winter*, 8–pp. IEEE.

Spivey, J. M. 1988. *Understanding Z: a specification language and its formal semantics*, Volume 3. Cambridge University Press.

Tofts, C., and G. Birtwistle. 1998. "A denotational semantics for a process-based simulation language". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8 (3): 281–305.

Tolk, A., S. Diallo, and C. Turnitsa. 2006. "Merging Protocols, Grammar, Representation, and Ontological Approaches in Support of C-BML". In *2006 Fall Simulation Interoperability Workshop*.

Tolk, A., S. Diallo, and C. Turnitsa. 2007. "A system view of C-BML". In *2007 Fall Simulation Interoperability Workshop*.

Tolk, A., S. Y. Diallo, J. J. Padilla, and H. Herencia-Zapana. 2013. "Reference modelling in support of M&Sfoundations and applications". *Journal of Simulation* 7 (2): 69–82.

Visser, E. 2008. "WebDSL: A case study in domain-specific language engineering". *Generative and Transformational Techniques in Software Engineering II*:291–373.

Woodcock, J., P. G. Larsen, J. Bicarregui, and J. Fitzgerald. 2009. "Formal methods: Practice and experience". *ACM Computing Surveys (CSUR)* 41 (4): 19.

AUTHOR BIOGRAPHIES

ROSS GORE Ross Gore holds a Doctorate of Philosophy (Ph.D.) and a Master's degree in Computer Science from the University of Virginia and a Bachelor's degree in Computer Science from the University of Richmond. He is working as a Post-Doctoral Researcher within the Modeling and Simulation Interoperability (MSI) Lab at The Virginia Modeling, Analysis and Simulation Center (VMASC) and will join the faculty of Gettysburg College in the Fall of 2013. His email address is rgore@odu.edu and his web page is http://www.vmasc.odu.edu/gore.html.

SAIKOU DIALLO is a Research Assistant Professor at the Virginia Modeling Analysis and Simulation Center (VMASC) of the Old Dominion University (ODU). He received his M.S. in Modeling & Simulation (2006) and his Ph.D. in M&S (2010) from ODU. His research focuses on the theory of interoperability as it relates to Model-based Data Engineering and Web Services for M&S applications. He is currently the co-chair of the Coalition Battle Management Language drafting group, an M&S IEEE standard development group. His e-mail and web addresses are sdiallo@odu.edu and http://www.vmasc.odu.edu/diallo.html, respectively.