Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

SIMULATION-SPECIFIC CHARACTERISTICS AND SOFTWARE REUSE

Joseph C. Carnahan
Paul F. Reynolds, Jr.
David C. Brogan

Modeling and Simulation Technology Research Initiative
Department of Computer Science, University of Virginia
Charlottesville, VA 22904-4740, U.S.A.

ABSTRACT

We argue that simulations possess interesting characteristics that facilitate adaptation. Simplifying assumptions, stochastic sampling, and event generation are common features which lend themselves to adaptation for reuse. In this paper, we explore simulation-specific characteristics amenable to adaptation and the ways they can be exploited in support of reuse. Our work is of particular relevance to research in component based simulations and dynamic data driven application systems, where adaptability and reuse are essential.

1 INTRODUCTION

Simulations are an important tool partly because they can be reconfigured and reused more cheaply than real-world experiments. Once a simulation is in use, running it on new data or with new parameters is usually just a matter of a few keystrokes or dragging and dropping a different file. However, there are still significant limitations on how simulations can be reused when requirements change.

Consider the challenge of building dynamic data-driven application systems (DDDAS). The DDDAS goal is to yield predictions more accurately and efficiently by processing experimental data in real-time and changing the software simulation appropriately (Darema 2004). The experimental data may indicate that the simulation should shift its region of exploration, change one or more of its starting assumptions, or adjust its level of resolution. The required changes are often more drastic than what can be accomplished just by tuning simulation parameters, because designers often do not know what types of changes will be suggested by the experimental data (Brogan et al. 2005).

The question is this: How can we rapidly make significant changes to simulation software when requirements change? Building flexible software has been a goal of the software engineering community for decades, and yet relatively few effective solutions have been found (Brooks 1995). Therefore, to make rapid simulation adaptation possible, we should look for approaches that exploit simulation-specific characteristics. Expanding on Bartholet et al. (2004), we note the following characteristics that are prevalent in simulations and yet relatively rare in other software:

- dependence on simplifying assumptions,
- importance of insight versus precision,
- use of stochastic sampling,
- event generation, and
- time management.

These characteristics can be exploited to speed up the process of simulation adaptation, which in turn reduces the cost of simulation reuse. In this paper, we discuss techniques that lead to new simulation adaptation technologies, and we compare these simulation-specific approaches to the state of the art in general-purpose software adaptation.

2 THE CHALLENGE OF SOFTWARE REUSE

In principle, the fastest way to build any kind of software is to reuse one or more components that have already been written (McIlroy 1968). However, it is often difficult to locate a component that does *exactly* what is needed to solve the problem at hand. Thus a solution generally entails taking an imperfectly suited component and adapting it to fit correctly. Hence, for simulations as well as for other software, the goal of effective reuse reduces to the problem of adapting components to fit a new scenario.

2.1 Design for Change

In the software engineering community, numerous advances have been made in designing software to support future changes. Major landmarks and recent developments in this area include

- structured programming (Dijkstra 1972),
- information hiding (Parnas 1972),

- program families (Parnas 1978),
- object oriented programming (Rentsch 1982),
- component architectures (Sullivan and Knight 1996),
- aspect oriented programming (Kiczales et al. 1997), and
- feature oriented programming (Batory et al. 2002).

These techniques demonstrate how to write software in such a way that future modifications require fewer changes to the code. They are not sufficient for automating the process of software adaptation. However, they make manual modifications easier and simplify software designs to the point where semi-automatic adaptation becomes a possibility. Hence, while these techniques do not provide for the degree of automation required for adaptable component-based and data-driven simulation systems, research into automated simulation adaptation could not proceed without them.

Other researchers have studied the role of software engineering in simulation. However, rather than seeking simulation-specific software engineering techniques, most have focused on making general-purpose software engineering principles accessible to simulation designers who lack formal training in computer programming (McKay et al. 1986). More recently, Bartholet et al. (2004) examined the relationship between component-based software engineering and simulation composability, concluding that while simulations are often dauntingly large and semantically complex components, simulation users can still benefit from component-based software technologies.

2.2 Automating Adaptation

Even using the best current software engineering practices, manual code changes must be made in order to adapt a program. Researchers have tried using optimization and other fully automated techniques to adapt software, but they have met with limited success outside of specific application domains. Genetic algorithms (GAs) are one example of automated adaptation. In GAs, variations of a program compete with each other to imitate a desired behavior. New versions of the program based on the best variations of this round are allowed to compete in the next round of development (Forrest 1996). However, techniques like GAs often produce poor solutions or suffer from performance problems, because they do not effectively exploit insight that developers may have about the problem the software is addressing.

In order to exploit insight about specific problems, scientists have investigated techniques targeted at particular classes of software. This has led to the development of program generators, which convert a description of a system in a domain-specific language to a working program (Smaragdakis, Huang, and Zook 2004). The development of domain-specific languages reduces the size and complexity of the code that needs to be changed to modify a program. Hence, program generators are a form of automation that accelerates software adaptation.

2.3 Specializing for Simulations

The simulation community was among the first to develop domain-specific languages. Early simulation languages included GPSS (Gordon 1978), SIMSCRIPT (Kiviat 1968), and Simula (Nygaard and Dahl 1978). These languages took advantage of the fact that simulations (especially discrete-event simulations) tend to contain certain common structures, such as

- random number generators,
- an event list, and
- interacting entities or processes.

By providing built-in commands for creating these structures, these languages accelerated simulation development. In effect, these languages addressed the challenge of simulation construction in the way that we are addressing simulation adaptation today.

Many newer simulation languages have been developed, improving on their predecessors by automating more of the common tasks associated with simulations or adding constructs to support parallel computation (Nance 1993, Bagrodia 1998). The issue of adaptation was first explicitly addressed in simulation languages with the appearance of visual development tools, where users are encouraged to use diagrams to define relationships among existing components.

2.4 Semi-Automated Simulation Adaptation

The ability to develop simulations rapidly is a good start, but additional work is needed to facilitate simulation adaptation. In this area, several techniques exist. Computational steering is one example, where a simulation and an expert user are connected through a real-time visualization and control interface. This allows the user to tune simulation parameters based on how the simulation is currently behaving (Gu, Vetter, and Schwan 1994; Parker and Johnson 1995).

Another example is COERCE, which is based on identifying flexible points in a simulation and applying a mixture of optimization and manual modifications to change a simulation's behavior (Reynolds 2002; Carnahan, Reynolds, and Brogan 2004a). This paper builds on COERCE by analyzing the underlying properties of simulations that make semi-automated adaptation possible.

Table 1: Summary of Simulation-Specific Characteristics

Characteristic	*Possible Applications*
Model abstraction assumptions	Flexible points, which document the assumptions' significance, possible alternatives, and constraints
Importance of insight versus precision	Opportunity to learn from the adaptation process itself or to accept intermediate results
Use of stochastic sampling	Switching distributions, tuning distribution parameters, and interchanging stochastics with detailed simulation
Event generation	New simulation representations and adaptation operations
Time management	New simulation representations and adaptation operations

3 SIMULATION-SPECIFIC CHARACTERISTICS

Simulations are used to address questions arising from issues of interest. To address these questions, simulations typically represent assumptions, create events (not necessarily discrete), maintain a notion of time, and employ stochastics to capture unknowns or processes that can only be characterized probabilistically. Time, event generation and stochastics present fruitful opportunities for exploring alternative approaches and answering questions about the issues of interest. As the questions change, opportunities for adaption occur in that flexibility built into simulations.

A number of simulation characteristics that contribute to flexibility are summarized in Table 1.

3.1 Model Abstraction Assumptions

One unavoidable step in building a simulation is deciding what *not* to model. Because each simulation only represents one part of the universe, all simulations contain simplifying assumptions. Examples include representing only certain classes of entities, a bounded region of space, or a finite length of time. Also, setting a simulation parameter as a constant precludes using other values for that parameter. Likewise, choosing one algorithm over another can be seen as a simplification of a more complex simulation that would have dynamically decided which perspective (and hence algorithm) to use for modeling a particular phenomenon.

Changing model abstraction assumptions may be necessary when a simulation is expanded or composed with another simulation. Model assumptions are a subset of simulation design decisions, which are in turn what existing software engineering techniques are designed to address (Parnas 1972). However, model abstraction assumptions often reflect real-world possibilities that could have been included in the model. Hence, our knowledge about the simulated phenomenon gives added insight into how these assumptions can be changed.

3.2 Importance of Insight versus Precision

A related characteristic of simulations is the role of approximation and insight in how simulations are used. The fact that simulations are usually incomplete representations of reality means that many are used more as a source of insight or trend analysis than as a predictor of exact results. Simulation users often go through an iterative process: They run simulations in order to gain insights that guide experimentation, later returning to the simulation and modifying it to match the experimental data. Hence, the *measures of effectiveness* for many simulations may emphasize factors other than precision (Zeigler, Praehofer, and Kim 2000). In fact, an incorrect simulation prediction can be beneficial when it reveals a problem with the experimenter's assumptions.

Obviously, many simulations provide precise and accurate results, but the acceptance of approximations or insight alone is not generally found in software other than simulations. Hence, there are situations where it would be difficult to adapt a piece of software to meet an exact numerical requirement but relatively easy to refine a simulation so that it can provide useful insight and analytical guidance. In addition, this means that using a partially-adapted simulation may provide guidance on how to complete the adaptation in ways that other partially-adapted software would not.

3.3 Stochastic Sampling

Simulations often employ stochastic sampling and random number generation. In many cases, random numbers represent simplifying assumptions, because a stochastic distribution is often used as an abstract model of a more complex process or as a stand-in for unknown information (Davis 2000). For instance, in a queuing model of bank tellers, the rate at which customers arrive may be regarded as Poisson. However, that stochastic process is replacing a simulation of each customer, who would actually make the decision about when to come to the bank based on a work schedule, family responsibilities, and other factors. Hence, it would be possible to replace the random arrival process with a more detailed simulation that models the daily routines of a pool of potential bank customers.

Stochastic processes are a special class of simplifying assumptions because they are relatively easy to analyze and modify. Many distributions have well-known shapes and properties, giving a good good idea of what to expect from switching distributions. Also, commonly-used distributions have relatively short lists of parameters that can be tuned by optimization tools to direct a simulation's behavior (Ólafsson and Kim 2002). Lastly, when taking a sample from a distribution often stands in for a more complex simulation, uses of random numbers in a simulation may be places where more detail could be provided. Hence, stochastic sampling provides a useful access point for adapting simulations.

3.4 Event Definition and Scheduling

Numerous software systems process "events" in some form, from graphical user interfaces responding to mouse clicks to device drivers responding to output completion events. However, simulations are interesting in that they not only *process* simulated events but also *generate* them. In fact, many simulations can be viewed as a sequence of event generation operations. The specifics of the generated events are candidates for adaptation to meet new requirements.

Simulations have been represented in terms of events, e.g. event graphs (Schruben 1983, Buss 1996). These approaches have focused on events at the time they occur rather than at the time they are generated and scheduled for future treatment. By focusing on generation time we believe there will be broader opportunities for exploiting adaptability.

3.5 Time Management

Another characteristic of simulations is the management of an internal concept of time. Not only is the nature of events dictated by a simulation, but timing and temporal relationships as well. Adapting a simulation may mean reordering existing events or adding and removing new events. By representing simulations as timelines with associated event generation operations, we can create a visual language for describing simulation behaviors that allows us to formally describe changes to those behaviors. Such a language would facilitate adaptation by making the temporal flexibility in simulations more apparent and by enabling static analysis of proposed adaptations.

4 APPLICATIONS

Given that typical simulations possess interesting characteristics, how can we exploit them to make simulations more usefully adaptable? There are several examples, including identifying flexible points, defining event generating languages, and representing simulations as timelines. In each example, a simulation-specific characteristic provides additional information or a new perspective for adapting simulations.

4.1 Flexible Points

Several software engineering methodologies center around design decisions. However, as we noted in Section 3.1, many of the decisions in simulation design reflect simplifying assumptions about the simulated phenomenon or its environment. As a result, a subject matter expert should understand these design decisions in terms of what they mean for the scope of the simulation and its expected behavior. This additional insight is useful for making future changes because it indicates likely changes and provides constraints on those changes.

To best exploit this insight, we define certain elements of the simulation to be *flexible points*:

Definition 1 *A flexible point is an element of a simulation that reflects an assumption about the simulated phenomenon.*

Flexible points are the basis for a technology called COERCE. COERCE consists of two parts, coercibility and coercion. In simulation coercion, a subject matter expert identifies the flexible points that relate to a desired change in a simulation's behavior (Carnahan, Reynolds, and Brogan 2003). Then, either optimization, manual modification, or both are used to find a new value for that flexible point (Waziruddin, Brogan, and Reynolds 2003). In coercible simulations, important flexible points are identified at design time and throughout the life cycle of a simulation. Specialized language constructs are inserted to describe each flexible point, its possible values, and any other relevant metadata (Carnahan, Reynolds, and Brogan 2004b). The relationship between coercibility and coercion is shown in Figure 1. By capturing information about the simplifying assumptions that went into the design of a simulation, it becomes possible to automate parts of the simulation adaptation process.

4.2 Simulations as Event-Generating Systems

As noted in Section 3.4, a distinguishing characteristic of simulations is the role of event generation. Rather than responding to a series of input events, often a simulation's job is to generate a series of events of its own. Hence, a simulation can be manipulated by not only changing how events are handled but how they are generated in the first place.

When a user needs to adapt a simulation to meet new requirements, the user may

- add or remove events from the simulation,
- change the behavior of an event (that is, change how an event is handled), or

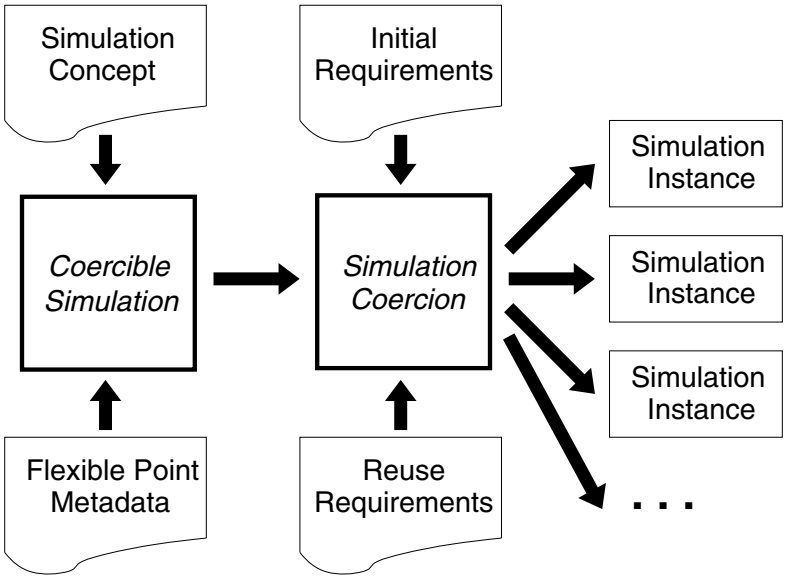

Figure 1: Life Cycle of Coercible Simulations

- change how events are generated.

The first two of these approaches could be applied to general-purpose software as well as simulations: Adding new events to a piece of software is analogous to extending it to handle new kinds of input, and changing the behavior of an event is equivalent to changing how the software responds to existing inputs. However, the idea of changing how events are generated does not make sense unless the system has a notion of generating and scheduling events.

Consider a model of a simple queuing system. Typically, such a system would be simulated by defining an event for job arrivals and an event for when a job completes service. This simulation could be adapted by adding or removing events, such as adding an event to represent machine breakdowns. Similarly, other kinds of software have event-handling routines, such as graphical user interfaces that can be adapted by making them sensitive to additional clicks and key presses. Also, this simulation could be adapted by changing how an event is handled, such as by modifying the arrival event to discard arrivals that occur when the queue size has exceeded some threshold. Other software can also be changed in how it responds to events, such as improving how an email program responds to new message arrivals by adding filters to block unwanted messages.

However, there is no analog in non-simulation software for changing event-generating activities. The simulation could change the timing with which new events are scheduled or even redefine the behavior of events as they are generated. For example, when an arrival event generates a service event, it signifies that the queue was empty at the time of arrival. Thus, the job could immediately enter service. When one service event is generated by another, it means the queue was not empty when the previous job was completed. To reflect this, the user could change the behavior of service events so that when they are generated by arrivals they do not behave the same as when they are generated by other service events, such as by adding a "warm-up time" to delay the completion of service events that follow an empty queue.

These kinds of modifications apply to simulations written in any language, but a simulation programming language could be extended to make this kind of modification even easier. For instance, a language might allow the user to add parameters that change how events behave depending on when and how they are generated. Still, even without such language support, there are approaches for adapting event-generating systems that do not apply to other classes of software.

4.3 Simulations as Timelines

Another opportunity arises from simulation's use of logical time. Because simulations represent events occurring in time, it is possible to depict simulation behavior as a (possibly repeating) timeline of events. Then, simulation adaptation can be thought of as a transformation from one timeline to another.

Consider a set of operations defined on timelines, such as adding, deleting, and reordering intervals, as well as merging and appending entire timelines. Even with a relatively small set of operations, it would be possible to describe any simulation adaptation in terms of a set of timeline operations: This can be proven by observing that looping and appending timelines is equivalent to using loops and sequential instructions in a programming language, which

in turn is sufficient to imitate any program that can be computed on a Turing machine (Böhm and Jacopini 1966).

Representing simulations as timelines offers several potential benefits:

1. Specifying adaptations in terms of well-defined timeline operations may be simpler and/or more precise than specifying the changes in terms of changes to the simulation code.
2. By changing the order of intervals and allowing operations to be applied repeatedly, even a very restricted set of timeline operations could produce the interesting variations that a user seeks.
3. The ability to formally describe an adaptation makes it possible to analyze the effects of different adaptations in relation to each other, such as verifying that certain simulation properties are always preserved.

5 DISCUSSION

Adapting programs to meet new requirements continues to be an important challenge both in simulation and in general software. However, by focusing on the characteristics that set simulations apart, we expect to find new ways to adapt simulations more quickly and effectively. Particularly with the growing demand for automation in simulation adaptation, such as for dynamic data-driven application systems, it is important to take advantage of these simulation-specific characteristics.

5.1 Future Work

Our approaches to exploiting simulation-specific characteristics for adaptation are evolving. We have created a prototype language for identifying flexible points and describing how they could be exploited, called the Flexpoint Markup Language (Flex ML). However, considerable work must be done to classify different types of flexible points, identify their utility, study how changing one flexible point affects others, and provide guidance on how to build coercible simulations.

Likewise, the simulation timeline representation is an idea with considerable room for expansion. Even with a basic set of operations defined on timelines, more complex operations may need to be defined in order to succinctly describe common adaptations or to facilitate formal analysis. As with designing any new language, a language of operations on timelines must consider issues of readability, orthogonality, and efficiency, with a particular concern about how different operations interact.

Lastly, there are likely to be other ways to exploit simulation-specific characteristics to represent simulations in more flexible ways. As noted in Section 4.2, a simulation language could be extended to make event generation more explicit and to simplify the task of changing how events are generated. Additional work would then be required to evaluate such a language extension and determine exactly how it affects simulation adaptability.

5.2 A Note on Simulations as White Boxes

One distinguishing characteristic that is often attributed to simulations is that they are "white boxes": Whereas users of other software only make use of their programs' outputs, users of simulation study the internal state of their simulations in order to gain insight (Davis et al. 2000, Davis and Anderson 2003, Zeigler, Praehofer, and Kim 2000, etc.). It is argued that this property makes simulation composition more difficult than using other kinds of component-based software, as simulations must not only agree at the interfaces but also in their internal assumptions.

However, we argue that this distinction is primarily a disagreement as to what constitutes the "interface" of a software component. Simulation components that agree in their data types but not in the assumptions underlying how those data are generated can not be validly composed. However, the same is true for many non-simulation software systems: The phenomenon of *architectural mismatch* (Garlan, Allen, and Ockerbloom 1995) refers to when any two software components can not be composed because of contradictory assumptions about the meaning of their data. To overcome architectural mismatch, software engineers have had to restrict composition to well-defined frameworks (Sullivan and Knight 1996) and to document the implicit as well as explicit assumptions as part of the software interface (Janicki, Parnas, and Zucker 1977, Bartussek and Parnas 1977, Parnas 1997). Following such practices makes designing simulation interfaces dauntingly complicated, as many previously hidden assumptions must now be included (Spiegel, Reynolds, and Brogan 2005). Still, because the challenge of identifying and formally documenting assumptions has haunted other domains of software as well as simulation, we do not include the importance of internal assumptions in our list of distinguishing characteristics.

That being said, adding hidden assumptions to the list of simulation-specific characteristics does not diminish the importance of the other five characteristics in building more adaptable simulations.

5.3 Impact

The goal of our work here is to discover ways to overcome the limitations of traditional software adaptation techniques on simulations. By taking advantage of both explicit identification of simulation assumptions and expert knowledge, automatic or semi-automatic discovery of alternatives for adapting an existing simulation becomes feasible. By de-

scribing simulation adaptations as operations on events and timelines, it becomes possible to analyze them in advance and verify that the system will remain stable through future data-driven adaptations. By exploring additional ways to represent simulations distinctly from other software, even more useful techniques for simulation adaptation can be discovered. Exploiting simulation-specific characteristics can accelerate adaptation and reduce the cost of simulation reuse.

ACKNOWLEDGMENTS

We gratefully acknowledge support from the DDDAS program at the National Science Foundation (ITR 0426971), as well as from our colleagues in the Modeling and Simulation Technology Research Initiative (MaSTRI) at UVA.

REFERENCES

Bagrodia, R. L. 1998. Parallel languages for discrete-event simulation. *Computational Science and Engineering* 5:27–38.

Bartholet, R. G., P. F. Reynolds, D. C. Brogan, and J. C. Carnahan. 2004. In search of the Philosopher's Stone: Simulation composability versus component-based software design. In *Proceedings of the 2004 Fall Simulation Interoperability Workshop*. Orlando, Florida: Simulation Interoperability Standards Organization.

Bartussek, W., and D. L. Parnas. 1977, Dec. Using assertions about traces to write abstract specifications for software modules. Technical Report TR77-012, University of North Carolina at Chapel Hill.

Batory, D., C. Johnson, B. MacDonald, and D. von Heeder. 2002. Achieving extensibility through product-lines and domain-specific languages: a case study. Volume 11, 191–214: ACM Press.

Böhm, C., and G. Jacopini. 1966. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM* 9 (5): 366–371.

Brogan, D. C., P. F. Reynolds, R. G. Bartholet, J. C. Carnahan, and Y. Loitière. 2005. Semi-automated simulation transformation for DDDAS. In *Computational Science - ICCS 2005: 5th International Conference*. Heidelberg, Germany: Springer-Verlag.

Brooks, F. 1995. *The mythical man-month*. Anniversary ed. Reading, MA: Addison-Wesley.

Buss, A. H. 1996. Modeling with event graphs. In *Proceedings of the 28th Winter Simulation Conference*, 153–160. New York, NY: ACM Press.

Carnahan, J. C., P. F. Reynolds, and D. C. Brogan. 2003. An experiment in simulation coercion. In *Proceedings of the 2004 Interservice/Industry Training, Simulation, and Education Conference*. Arlington, Virginia: National Training Systems Association.

Carnahan, J. C., P. F. Reynolds, and D. C. Brogan. 2004a. Language support for identifying flexible points in coercible simulations. In *Proceedings of the 2004 Fall Simulation Interoperability Workshop*. Orlando, Florida: Simulation Interoperability Standards Organization.

Carnahan, J. C., P. F. Reynolds, and D. C. Brogan. 2004b. Visualizing coercible simulations. In *Proceedings of the 2004 Winter Simulation Conference*, ed. R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, 411–419. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Darema, F. 2004. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *Computational Science - ICCS 2004: 4th International Conference*, ed. M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Volume 3038. Heidelberg, Germany: Springer-Verlag.

Davis, P. C., P. A. Fishwick, C. M. Overstreet, and C. D. Pegden. 2000. Model composability as a research investment: responses to the featured paper. In *WSC '00: Proceedings of the 32nd Winter simulation conference*, 1585–1591: Society for Computer Simulation International.

Davis, P. K. 2000. Exploratory analysis enabled by multiresolultion, multiperspective modeling. In *WSC '00: Proceedings of the 32nd Winter simulation conference*, 293–302: Society for Computer Simulation International.

Davis, P. K., and R. H. Anderson. 2003. Improving the composability of Department of Defense models and simulations. Technical Report MG-101, RAND National Defense Research Institute, Santa Monica, CA.

Dijkstra, E. W. 1972. The humble programmer. *Communications of the ACM* 15 (10): 859–866.

Forrest, S. 1996. Genetic algorithms. *ACM Computing Surveys* 28 (1): 77–80.

Garlan, D., R. Allen, and J. Ockerbloom. 1995. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, 179–185: ACM Press.

Gordon, G. 1978. The development of the general purpose simulation system (GPSS). In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, 183–198: ACM Press.

Gu, W., J. Vetter, and K. Schwan. 1994. An annotated bibliography of interactive program steering. *SIGPLAN Not.* 29 (9): 140–148.

Janicki, R., D. L. Parnas, and J. Zucker. 1977. Tabular representations in relational documents. In *Relational methods in computer science*, ed. C. Brink and G. Schmidt, Chapter 12, 184–196. Springer Verlag.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-

oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, ed. M. Aksit and S. Matsuoka, Volume 1241, 220–242. Heidelberg, Germany: Springer-Verlag.

Kiviat, P. J. 1968. Introduction to the SIMSCRIPT II programming language. In *Proceedings of the second conference on Applications of simulations*, 32–36: Winter Simulation Conference.

McIlroy, M. D. 1968. Mass-produced software components. In *NATO Science Committee Conference on Software Engineering*, ed. P. Naur and B. Randell, 138–150. Brussels, Belgium: NATO Scientific Affairs Division.

McKay, K. N., J. A. Buzacott, J. B. Moore, and C. J. Strang. 1986. Software engineering applied to discrete event simulations. In *WSC '86: Proceedings of the 18th Winter simulation conference*, 485–493: ACM Press.

Nance, R. E. 1993. A history of discrete event simulation programming languages. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, 149–175: ACM Press.

Nygaard, K., and O.-J. Dahl. 1978. The development of the Simula languages. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, 245–272: ACM Press.

Ólafsson, S., and J. Kim. 2002. Simulation optimization. In *WSC '02: Proceedings of the 34th Winter simulation conference*, 79–84: Winter Simulation Conference.

Parker, S. G., and C. R. Johnson. 1995. SCIRun: a scientific programming environment for computational steering. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, 52.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15 (12): 1053–1058.

Parnas, D. L. 1978. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, 264–277: IEEE Press.

Parnas, D. L. 1997. Precise description and specification of software. In *Mathematics of dependable systems II*, ed. V. Stavridou, 1–14. Clarendon Press.

Rentsch, T. 1982. Object oriented programming. *SIGPLAN Notices* 17 (9): 51–57.

Reynolds, P. F. 2002. Using space-time constraints to guide model interoperability. In *Proceedings of the 2002 Spring Simulation Interoperability Workshop*. Orlando, Florida: Simulation Interoperability Standards Organization.

Schruben, L. 1983. Simulation modeling with event graphs. Volume 26, 957–963: ACM Press.

Smaragdakis, Y., S. S. Huang, and D. Zook. 2004. Program generators and the tools to make them. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 92–100: ACM Press.

Spiegel, M., P. F. Reynolds, and D. C. Brogan. 2005. A case study of model context for simulation composability and reusability. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Sullivan, K. J., and J. C. Knight. 1996. Experience assessing an architectural approach to large-scale systematic reuse. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, 220–229: IEEE Computer Society.

Waziruddin, S., D. C. Brogan, and P. F. Reynolds. 2003. The process for coercing simulations. In *Proceedings of the 2003 Fall Simulation Interoperability Workshop*. Orlando, Florida: Simulation Interoperability Standards Organization.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation*. 2nd ed. San Diego, CA: Academic Press.

AUTHOR BIOGRAPHIES

JOSEPH C. CARNAHAN is a Ph.D. candidate in Computer Science and a member of MaSTRI at the University of Virginia. Joseph earned his B.S. in Computer Science at the College of William and Mary, and has held the position of Scientist at the Naval Surface Warfare Center, Dahlgren Division. His email address is <carnahan@virginia.edu>.

PAUL F. REYNOLDS, JR. is a Professor of Computer Science and a member of MaSTRI at the University of Virginia. He has conducted research in Modeling and Simulation for over 25 years, and has published on a variety of M&S topics, including parallel and distributed simulation, multi-resolution modeling and coercible simulations. He has advised industrial and government agencies on matters relating to modeling and simulation. He is a plank holder in the DoD High Level Architecture. His email address is <reynolds@virginia.edu>.

DAVID C. BROGAN earned his Ph.D. from Georgia Tech and is currently an Assistant Professor of Computer Science and a member of MaSTRI at the University of Virginia. For more than a decade, he has studied simulation, control, and computer graphics for the purpose of creating immersive environments, training simulators, and engineering tools. His research interests extend to artificial intelligence, optimization, and physical simulation. His email address is <brogan@virginia.edu>.