An End-to-End Domain Specific Modeling and Analysis Platform

Arman Shahbazian George Edwards Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA, USA 90089
{armansha, gedwards, neno}@usc.edu

ABSTRACT

Software architecture models are specifications of the principal design decisions about a software system that primarily govern its structure, behavior, and quality. They serve as a basis for experimentation and rationalization of design decisions. While many techniques utilize and support software architecture modeling and analysis, a recurring obstacle is that often advances in one area (e.g., architecture-based modeling) tend to be disconnected from those in another area (e.g., simulation). In this work we aim to provide an end-to-end model-driven engineering approach, called DomainPro, that bridges this gap and supports engineers throughout the software modeling, analysis, and implementation process by automatically synthesizing a range of *model interpreters* (*MI*). DomainPro is also available to download at https://goo.gl/4sRT9B. You can also watch the demo video describing DomainPro's prominent features at https://youtu.be/6rg7pC6bhO0.

1. INTRODUCTION

Software architecture models are specifications of the principal design decisions about a software system that primarily govern its structure, behavior, and quality. Architecture models provide a blueprint for how the system is implemented, serve as a basis for experimentation with and rationalization of design decisions [11], and enable the automation of software engineering tasks. While new techniques regularly emerge, advances in one area (e.g., software modeling) tend to be disconnected from those in another area (e.g., simulation) [5]. As a result, great ideas are regularly thrown into the dustbin, only to be rediscovered later. Model-driven engineering (MDE) provides an opportunity to break this cycle. MDE allows software engineers to easily define and use special-purpose design abstractions called domain-specific languages (DSLs). DSLs are able to concisely and intuitively express software architecture models because they natively include the design abstractions that are most useful and natural for the system under development. In this paper we describe an end-to-end approach and the accompanying tool, called DomainPro, that enables engineers to automatically synthesize model interpreters for DSLs. The main ingredients of our approach are (1) the metamodel, (2) the metamodel interpreter, and (3) the model interpreters.

MiSE'16, May 16-17, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2896982.2896994

2. OVERVIEW

The over-arching goal of DomainPro is to simplify, and automate the development of software systems. It is intended for software architecture-based modeling, analysis, and code generation. We focus specifically on leveraging DSLs to automatically generate a range of tools that will be used to manipulate (i.e., "interpret") the application specific software models captured in the DSL. These tools support engineers in automatic (1) model editing, (2) model analysis and simulation, and (3) system implementation. DomainPro aims to maximize the flexibility in both presentation and analysis. The drawing environment enables engineers to attach semantics to different part of the drawing they have in mind. It does not restrict engineers to adhere to a certain notation or formalization.

Automated design analysis and code generation is achieved through model transformation. In practical terms a model transformation is usually implemented by a program called a *model interpreter* (MI). An MI reads the data contained in models, manipulates that data, and produces output for a particular semantic domain. The semantic domain may be implemented by an external application (e.g., an analysis tool) or run-time environment (e.g., middleware platform). The remainder of this section first describes the process of using DomainPro, then details its metamodeling facilities, model interpretation components, and domain-specific model analysis using the provided simulation MI.

Software engineers first use DomainPro's metamodel editor to create a metamodel that defines a DSL. The metamodel specification process consists of instantiating the types available in the metalanguage (the metatypes) and setting the values of their properties. DomainPro first invokes the appropriate metainterpreter, which produces a *model interpreter framework* (MIF) extension (a set of C# plug-in classes) by deriving the simulation or implementation semantics of the DSL types in the metamodel. DomainPro then compiles the provided MIF (also implemented in C#) with the extension. The output of the compilation is a domain-specific model editor, simulation generator, and C#

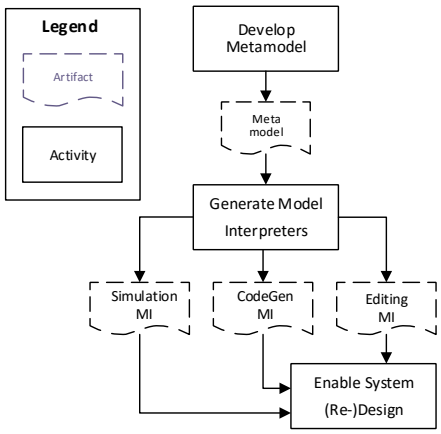


Figure 1: Software development process supported by DomainPro.

(.Net framework) code generator, already configured with the DSL's custom semantics.

DomainPro is implemented using Microsoft .Net Framework. It contains 4 sub-projects, *Core*, *Language Builder*, *Designer*, and *Analyst*. Language Builder is the metainterpreter, Designer is the editor MI, and Analyst is the simulation MI. By separating the core types and putting them in the Core project, we facilitate the isomorphic implementation of different MIs.Software development process that is supported by DomainPro is displayed in Figure 1

2.1 Metamodeling

Metalanguages are generally centered around a small set of basic metatypes, which we refer to as the *core* metatypes. The core metatypes are derived from basic information representation paradigms, such as the object-oriented or the relational data model. DomainPro targets software architecture-based modeling, analysis, and code generation. Therefore, the metalanguage should be sufficiently flexible to capture the range of abstractions and patterns commonly used for modeling software and systems architectures. Based on our literature review we identified the following: `structure`, `component`, `resource`, `interface`, `link`, `implementation`, `method`, and `datatype`. Once a set of core metatypes has been chosen, the next task will be to attach partial semantics to each metatype. Crucially, the semantics of metatypes should only incorporate assumptions shared among a broad family of DSLs. Semantics that vary from one application context to another will be left for engineers to specify using properties, as described below. Note that semantic assumptions will be incorporated directly into DomainPro's interpretation components, simplifying their implementations and increasing the scalability and efficiency of the MDE platform.

The use of semantic assumptions results in a trade-off between the ability to synthesize supporting toolsets and DSL flexibility. For example, current MDE platforms can synthesize model editors because their metatype semantics include visualization, presentation, and editing concerns, but they lack the semantics necessary to synthesize other types of tools. One approach for arriving at a set of semantic assumptions is co-refinement [8]. Co-refinement begins with an initial candidate set of metalanguage semantics (e.g., the semantics reflecting the well-understood constructs and abstractions underlying architecture-based software development [17]) and target platform semantics (e.g., the semantics of a programming language or middleware platform selected to implement the architectures). Expanding the metalanguage semantics strengthens restrictions on DSL definition, but weakens restrictions on automated model interpreter generation; expanding the semantics of the target platform (if possible) has the inverse result. Thus, through an iterative process, the semantics of each can be brought into alignment. In addition to embedded semantic assumptions, DomainPro attaches semantics to metatypes through properties. Properties are typed attributes and associations with other metatypes. In contrast to semantic assumptions, metatype properties are used to capture domain-specific semantics that vary from one context to another. This use of properties has the advantage that metamodel developers are not required to write intricate formal semantic specifications using a complex notation such as Structured Operational Semantics [1, 14]. Instead, software engineers can configure semantics using menus and dialogs, and automatically check properties for consistency. For example, the rendering of domain-specific types within the model editors of today's MDE platforms does not need to be specified because their rendering semantics are encoded within the properties of their metatypes. The flip side, in this case, is that the semantics that can be captured are restricted by the MDE platform's developer.

2.2 Interpretation

In contrast to the canonical architecture, each interpretation capability in DomainPro is realized through a paired *metainterpreter* and its associated *model interpreter framework* (*MIF*), which is the template for constructing the desired collection of editing, analysis, and code generation model interpreters (MIs).

Notionally, a MIF can be thought of as a virtual machine whose instruction set is the set of implemented transformation operations supplied by the metainterpreter, which is, in turn, akin to a compiler whose function is to generate programs to be executed by the MIF virtual machine. Our implementation of a MIF is like an application framework. This ensures that the MIF achieves requisite performance and scalability by internalizing program control logic and invoking application extensions, rather than being passively invoked by application control logic (*inversion of control*). Moreover, an application framework strictly controls the ways in which its behavior is modified, and thereby ensures that assumptions (including embedded semantic assumptions) are not violated.

The editing components of DomainPro operates similarly to existing MDE platforms [6]. However, other interpretation components of our approach are not present in existing MDE platforms. DomainPro's metainterpreter derives the semantics of domain-specific types from their metatype property definitions and determines a set of rules for transforming each domain-specific type to the target semantic domain—software architecture-based development. The metainterpreter encodes the transformation rules in an MIF extension. The resulting MIF implements the actual transformation logic (e.g., operations and algorithms) for each MI.

2.3 Simulation

As described DomainPro automatically generates fully configured, domain-specific interpreters that today have to be programmed manually. One type of interpreter generated by DomainPro is simulation generators. The simulation generator designed for DomainPro is a component-based variant of the widely used discrete event simulation paradigm [18]. DomainPro's simulation generators consist of the simulation MIF (built into DomainPro) and domain-specific simulation MIF extension code (autogenerated by DomainPro). Engineers specify a list of *watched types* to capture the behaviors of the system. Depending on the metatype of the watched types DomainPro attaches a fitting listener to that type that monitors and records the changes of its relevant attributes during the simulation time. Both of these are extensible and engineers can fine-tune these attributes, and define custom listeners to monitor other types if they need. Out of the box DomainPro provides listeners for the following types (monitored attributes are listed in the parentheses): Data (value), method (number of invocations, invocations intervals, execution time, blocking time), resource (idle capacity, queue length), and component (number of blocking methods, number of executing methods). In the next section we will look at an example to illustrate DomainPro's capabilities.

3. AN EXAMPLE

As an example to demonstrate first-hand how DomainPro works, we built a system with a distributed architecture based on a previously published description [2]. These types of systems are a fairly complex and representative showcase for DomainPro. Many globally renowned systems have similar architecture, like Skype (peer-to-peer video streaming), and Freenet (distributed file system). We focused on a specific subset of these systems, i.e., *distributed computation architectures* (DCAs). DCAs are used to solve massive problems by deploying highly parallelizable computations (i.e., sets of independent tasks) to dynamic networks of potentially faulty computing nodes. We used the specifications in [2] to design our system. Widely known and successful DCAs include MapReduce systems (e.g., Hadoop), grid systems (e.g., Globus), and volunteer computing systems (e.g., BOINC).

Since these systems use untrusted participants and interact over untrusted networks, they utilize redundancy to achieve acceptable levels of reliability. With redundancy, each task will be executed as several identical jobs on distinct nodes. There are several strategies to approach this, traditional redundancy, progressive redundancy, and iterative redundancy. We use iterative redundancy in our design, which distributes the minimum number of jobs required to achieve a desired confidence level in the result, assuming that all the jobs' results agree. However, if some results disagree, the confidence level associated with the majority result is diminished because of the chance that the disagreeing results are correct. In other words, the apparent risk of failure of the task is increased. The algorithm then reevaluates the situation and distributes the minimum number of additional jobs that would achieve the desired level of confidence. This process repeats until the agreeing results sufficiently outnumber the disagreeing results to reach the confidence threshold. Figure 2 is adopted from [3].

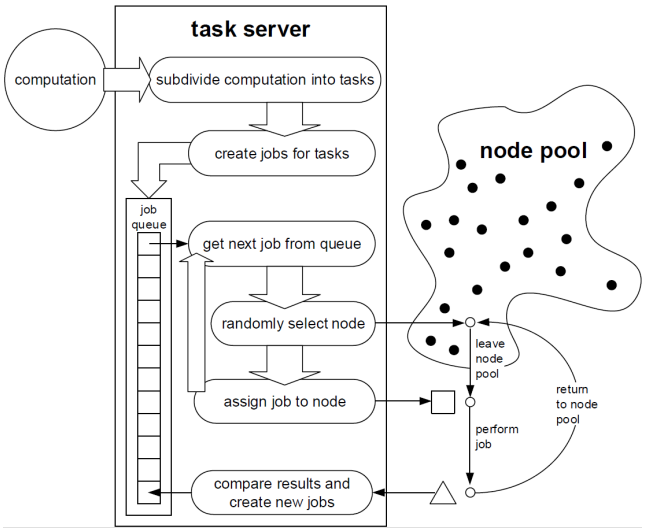


Figure 2: Conceptual model of our example system.

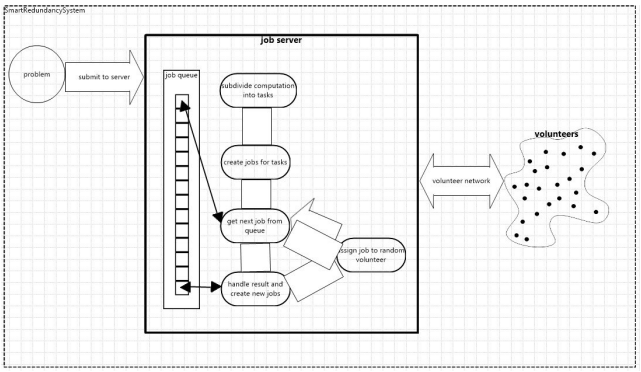


Figure 3: Our running example as modeled in DomainPro.

In this model we use the following nomenclature. A *computation* is the typically large problem being solved by a DCA. A *task* is one of the parts of the computation that can be performed independently of the others. A *job* is an instance of a task that a particular node performs. With redundancy, each task will be executed as several identical jobs on distinct nodes. In our model of a DCA, a *task server* breaks up a computation into a large number of tasks. The task server then assigns jobs to *nodes* in a node pool, ensuring that each node is chosen at random. After returning a response to a job to the task server, each node rejoins the node pool and can again be selected and assigned a new job. New volunteer nodes may join the pool while other nodes may leave.

In order to model this system in DomainPro we start from capturing the semantics of the domain in the metamodel. Figure 4 displays the metamodel of this system as defined in DomainPro. *ProblemSpec* captures the characteristics of the computation displayed in Figure 2, and *VolunteerPool* is the pool of nodes available in the system. We identified 4 domain-specific properties for our system. Each one of them determines how the system behaves and in turn affects its non-functional properties. These domain-specific

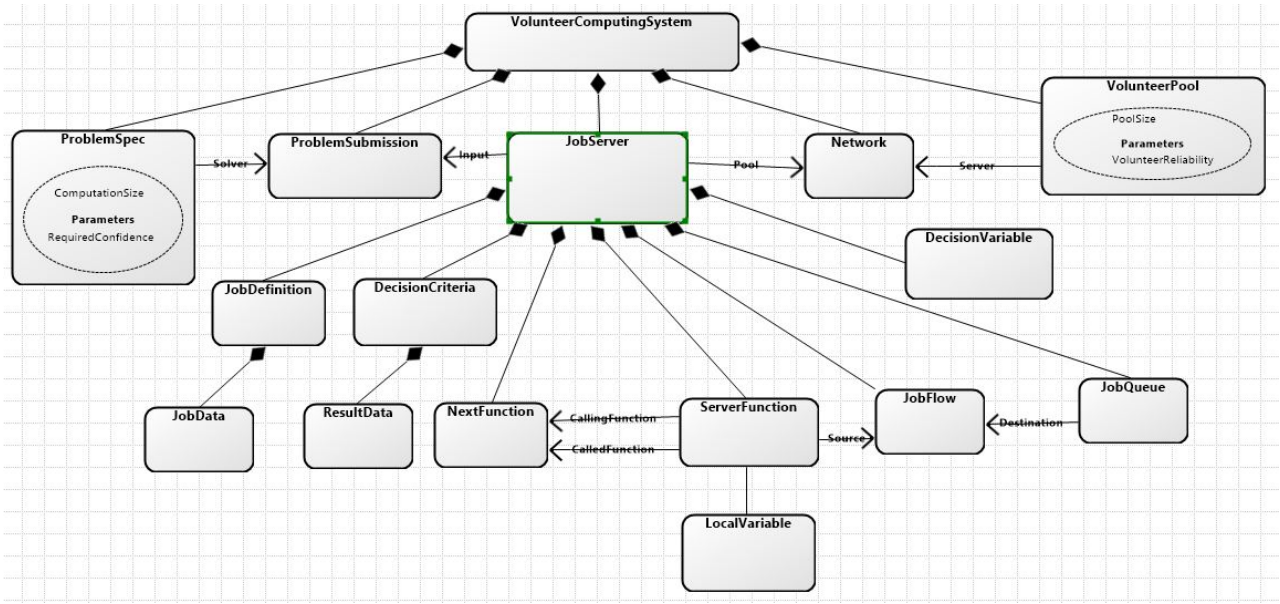


Figure 4: Metamodel of the running example in DomainPro.

properties are: (1) *ComputationSize*, which is the number of tasks required to complete the computation and essentially represents how big the computation is; *RequiredConfidence* is an integer representing the statistical confidence that we want to have in each result [1]; it is an integer number that shows the least number of nodes that have to return the same result for identical jobs (based on iterative redundancy principles it is the difference between the number of identical results and the rest of the results) (3) *NodeReliability*, that shows the probability of returning the correct result from each node; and (4) *PoolSize*, is the number of available volunteer nodes. By defining these domain-specific parameters in the metamodel, engineers will be able to easily analyze the system's behavior by changing their values. This can be done either at the model, or while analyzing the system.

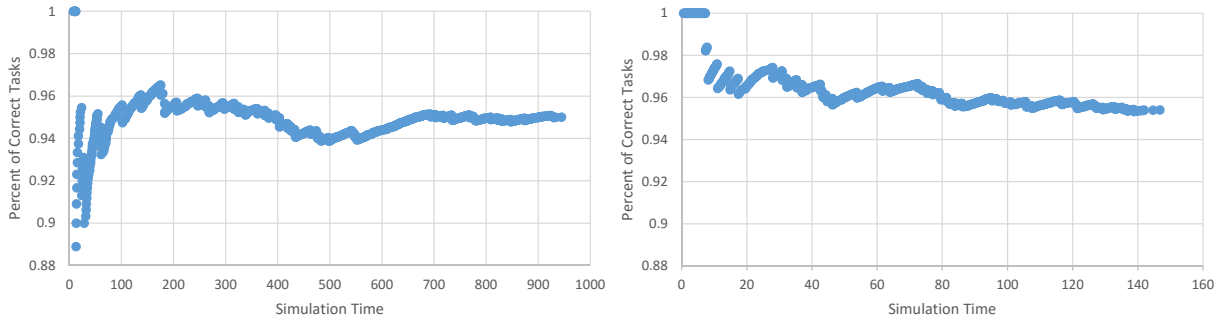
After defining the metatypes, their roles, and relationships to other metatypes, we can model the system. Figure 3 displays screen shots of this model created in DomainPro. As can be seen this model is quite similar to the original model of the system displayed in Figure 2. DomainPro's ability and flexibility in decoupling the presentation from semantics enables us to design our models in an already familiar manner. Assigning different values to domain-specific properties of this model can change the system's behavior drastically. For brevity we will only look at two quality attributes for this model: (1) *reliability* of the results, which is depicted by the percentage of tasks correctly computed; and (2) *efficiency* of computation, represented by the total number of generated jobs to complete the computation. These two quality attributes are monitored through the described watched types during the simulation of the system and their values at different times are recorded.

Intuitively it is clear that the described semantic properties are immensely intertwined and have conflicting effects on the quality attributes (e.g., higher *RequiredConfidence* increases the reliability while decreasing the efficiency). But to what extent different values for each one of the semantic properties affect the quality attributes is by no means clear. Figure 5 illustrates how DomainPro helps engineers overcome such conundrums by simulating the behavior of the system and analyzing the outcomes. We simulated the system using two different configurations. In both configurations we used the same value for the *PoolSize* (20 nodes), and the *ComputationSize* (1000 tasks). However 5(b) uses more reliable nodes, with less required confidence compared to 5(a). Results of the simulation show that although both project similar percent of correct completed tasks (*reliability*), but option 5(a) has generated almost 6 times the number of jobs 5(b) has generated. In other words, it's 6 times less *efficient*. This is a simple trade-off scenario between only two quality attributes, but if other non-functional aspects of the system, such as cost of acquiring nodes, are put in the mix, engineers will be facing more abstruse situations.

4. CONCLUSION

The goal of the work described in this paper is to simplify and accelerate the development of domain-specific model interpreters for software architecture quality analysis. To that end we designed and implemented a new approach called DomainPro. Our approach flexibility enables engineers to use the notations they are familiar with in designing their system. Engineers can start the design process in practically any drawing tool, and won't have to worry about having to shoehorn their design into a modeling framework with proprietary set of notations. DomainPro automatically synthesizes their models and generates the required analysis and simulation tool. This enables easy analysis of quality aspects of a model. In future we aim to expand this work in several ways. First, by enabling engineers to automatically optimize their system models based on a set of defined quality attributes. Second, we plan to incorporate *DEvA*

[1]Consider c as the value of the required confidence, when the number of jobs agreeing on a result are c units more than the rest of the jobs, then that job is marked complete.



(a) C=8, R=0.6, and G=36684

(b) C=3, R=0.75, and G=5630

Figure 5: Simulation results for two different configurations of the DCA system. C = *required confidence*, R = *node reliability*, and G = *number of generated jobs*. Vertical axis displays the percent of correct tasks completed until time t in the simulation, and horizontal axis depicts the simulation time.

[15] to discover possible event races and anomalies at the design level. Finally, DomainPro can be integrated with approaches like *Extract* [10] to enable analysis of existing systems for future development.

5. RELATED WORK

This section touches upon the most closely related work in two subcategories of MDE platforms: production platforms and experimental platforms.

Production MDE Platforms are in use by a large, active community of engineers. These platforms are feature-rich, robust, and mature, but they tend to lack the capabilities newly invented by researchers. Examples are Generic Modeling Environment (GME) [7], the Eclipse Graphical Modeling Framework (GMF) family of tools [4], and the MetaEdit+ Workbench [13].

Experimental MDE Platforms are primarily developed in academic research settings. DUALLY [12] is one of these platforms which eliminates the need to manually program model transformations. However, it still requires defining the mappings between languages, and exacerbates the model traceability problem of relating analysis results or bugs in generated code to the source architecture. The ALFAMA DSL workbench [16] automates the construction of DSLs and code generators for application frameworks. However, inferring DSLs from framework hot-spots tightly couples the DSL to a particular framework. model.

References

[1] L. Aceto et al. Structural operational semantics. *BRICS Report Series*, 6(30), 1999.

[2] Y. Brun et al. Smart redundancy for distributed computation. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 665–676. IEEE, 2011.

[3] Y. Brun et al. Self-adapting reliability in distributed software systems. 41:764 – 780, 2015.

[4] Eclipse Graphical Modeling Framework Project. www.eclipse.org/gmf/.

[5] G. Edwards et al. Automated analysis and code generation for domain-specific models. In *Working IEEE/IFIP Conference in Software Architecture, IEEE, 2012*, pages 161–170. IEEE, 2012.

[6] M. Garzon et al. Umple: A framework for model driven development of object-oriented systems. In *Software Analysis, Evolution and Reengineering (SANER), IEEE, 2015*, pages 494–498. IEEE, 2015.

[7] The Generic Modeling Environment. www.isis.vanderbilt.edu/Projects/gme/.

[8] S. A. Hissam et al. Packaging Predictable Assembly. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD)*, pages 108–124, Berlin, Germany, June 2002.

[9] J.-M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. In *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer, 2011.

[10] M. Langhammer et al. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2016.

[11] D. M. Le et al. An empirical study of architectural change in open-source software systems. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 235–245. IEEE, 2015.

[12] I. Malavolta et al. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. *IEEE Transactions on Software Engineering*, 36(1):119–140, January 2010.

[13] MetaEdit+ Workbench. www.metacase.com/mwb/.

[14] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[15] G. Safi et al. Detecting event anomalies in event-based systems. In *Fundamentals of Software Engineering (ESEC/FSE), 2015*. IEEE, 2015.

[16] A. L. Santos et al. Automating the Construction of Domain-Specific Modeling Languages for Object-Oriented Frameworks. *Journal of Systems and Software*, 83(7):1078–1093, July 2010.

[17] R. N. Taylor et al. *Software Architecture: Foundations, Theory and Practice*. Wiley Publishing, 2009.

[18] B. Zeigler et al. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.