The Split System Approach to Managing Time in Simulations of Hybrid Systems having Continuous and Discrete Event Components [*]

James Nutaro, Phani Teja Kuruganti,
Vladimir Protopopescu, Mallikarjun Shankar

Abstract

The efficient and accurate management of time in simulations of hybrid models is an outstanding engineering problem. General *a priori* knowledge about the dynamic behavior of the hybrid system (i.e., essentially continuous, essentially discrete, or "truly hybrid") facilitates this task. Indeed, for essentially discrete and essentially continuous systems, existing software packages can be conveniently used to perform quite sophisticated and satisfactory simulations. The situation is different for "truly hybrid" systems, for which direct application of existing software packages results in a lengthy design process, cumbersome software assemblies, inaccurate results, or some combination of these *independent of the designer's a priori knowledge about the system's structure and behavior.*

The main goal of this paper is to provide a methodology whereby simulation designers can use *a priori* knowledge about the hybrid model's structure to build a straightforward, efficient, and accurate simulator *with existing software packages*. The proposed methodology is based on a formal decomposition and re-articulation of the hybrid system; this is the main theoretical result of the paper. To set the result in the right perspective, we briefly review the essentially continuous and essentially discrete approaches, which are illustrated with typical examples. Then we present our new, *split system approach*, first in a general formal context, then in three more specific guises that reflect the viewpoints of three main communities of hybrid system researchers and practitioners. For each of these variants we indicate an implementation path. Our approach is demonstrated with an archetypal problem of power grid control.

[*]Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The submitted manuscript has been authored by a contractor of the U.S. Government under Contract DE-AC05-00OR22725. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

1 Introduction

A major technical problem encountered when building simulators that combine discrete event and continuous components is to precisely and efficiently align in time three discrete elements: i) the points of time at which solutions to the model's continuous equations are calculated; ii) events that are contingent on the continuous state variables; and iii) events that are contingent only on discrete variables. Solutions to this problem are numerous, but this plurality is due in large part to each solution being focused on a specific problem or small class of problems. In this paper, we discuss two types of solutions that are representative of the majority of extent simulation methods for combined systems. By merging specific aspects of these two methods we create a third that mitigates their greatest weaknesses and, at the same time, addresses a greater range of problems than either method alone.

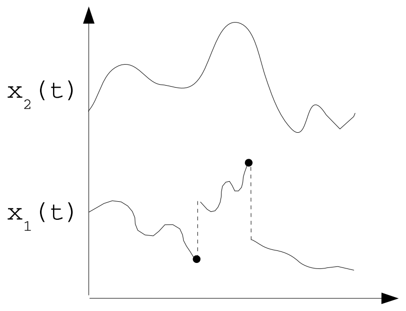
We begin by reviewing what we consider to be essentially continuous and essentially discrete approaches to managing time in hybrid simulations, illustrating these with typical examples of models that extent simulation tools handle well. Then we present our new approach, first in a general context that defines the hybrid simulation problem, followed by three more-specific guises that reflect three popular schemes for describing hybrid models. We indicate an implementation path for each of these variants. Finally, we illustrate our approach on an archetypal problem of power grid control.

The paper is structured as follows. In Section 2 we present a rough classification of the hybrid systems that existing tools aim to simulate: mostly continuous, mostly discrete, and a third class that is neither mostly discrete nor mostly continuous and hereafter will be called truly hybrid. In Sections 3 and 4 we discuss the two widely used approaches to simulating hybrid systems, namely Continuous Systems Simulation Languages (CSSL) and Combined Discrete Event/Continuous Simulation packages. These can be effectively used to simulate mostly continuous and mostly discrete systems, respectively, but they are not suited to simulating truly hybrid systems. Section 5 introduces our split hybrid system approach that integrates these existing approaches into a comprehensive simulation capability. Section 6 summarizes and concludes the paper.

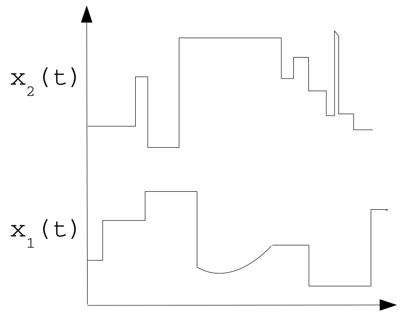
2 Three Main Classes of Hybrid Dynamics

The notion of a hybrid system is encompassed by a broad range of modeling formalisms. The relative scope and complexity of a model's discrete and continuous dynamics favors a particular choice of modeling formalisms. This in turn determines the selection of simulation tools.

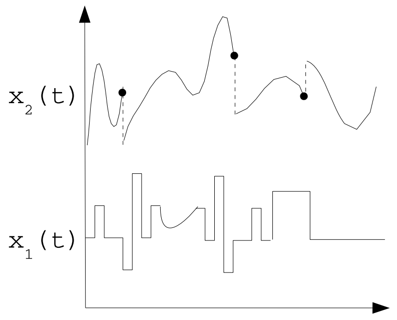
A clear-cut, unambiguous classification of hybrid systems is complicated and beyond the scope of this study. We shall limit ourselves to a rough, but intuitively appealing and practically useful, taxonomy whose main goal is to guide the selection of appropriate simulation technologies for hybrid systems. When judging the suitability of a technology, one has to weigh several factors that include (i) the conceptual simplicity and parcity of the underlying model(s), (ii) the effort required to construct the simulation software, with special emphasis on using existing, off-the-shelf simulation packages, (iii) the accuracy of the expected results, and (iv) the actual running time of the simulation on the available hardware.



(a) Type 1



(b) Type 2



(c) Type 3

Figure 1: Typical trajectories for the three classes of hybrid systems.

At a very cursory, but intuitively appealing, level our classification divides hybrid systems into (1) mostly continuous, (2) mostly discrete, and (3) truly hybrid. A more precise characterization reads as follows:

1. Type 1 systems with continuous trajectories that have or are interrupted by rare events,

2. Type 2 systems with essentially discrete event dynamics, possibly with rare intermittent intervals of continuous behavior, and

3. Type 3 hybrid systems with significant, interacting continuous and discrete dynamics.

These three types of systems create distinguishable trajectories, as illustrated in Fig. 1.

Systems of Type 1 have continuous, typically non-constant, trajectories that are interrupted by occasional discontinuities (discrete events), and these events are triggered by threshold crossings. More precisely, when the continuous trajectory of the system reaches a certain (threshold) value, it jumps instantaneously and discontinuously to a new value (state). Occasionally, the threshold values are known in advance, but typically they become available only as the solution progresses. In other words, an event is detectable only when it actually occurs[1].

Event locations in time are determined by the roots t_i of a set of equations $f_i(t) = 0$, where $i = 1, ..., n$. In general, the continuous functions $f_i(t)$ are written in terms of the model's time dependent, continuous state variables, i.e., $f_i(t) = f_i(x_1(t), ..., x_m(t))$. In this way, the time localization of discrete events and the construction of the continuous solution between events are intimately connected.

Systems of Type 2 have essentially discrete dynamics with rare intermittent stretches of non-constant continuous behavior. The discrete events are generated at times t_i, $i = 1, 2, ...$, that are determined recursively. The system state is unchanging (constant trajectory) between events. A state transition function Δ transforms the current state into a new state at times dictated by a time advance function ta. Denoting the system state variables by x, the general form of the discrete event evolution is

$$\begin{aligned} x_{n+1} &= \Delta(x_n) \\ t_{n+1} &= t_n + ta(x_n) \ . \end{aligned} \tag{1}$$

This formulation encompasses the DEVS formalism and most other discrete dynamic modeling frameworks (see, e.g., [1–3]). It is important to note that the state variables can be complex structures such as lists, queues, sets, etc., and that the dynamics described by Eqn. 1 can be sparingly interrupted by continuous evolutions.

Systems of Type 3 are truly hybrid systems; they are at the core of the approach we advance in this paper. These systems consist of multiple continuous and discrete components that are strongly interconnected. Discrete behaviors are described by both recursively generated event sequences and functions that depend on continuously evolving variables.

Due to the strong intertwining of continuous dynamics and discrete events, the mathematical formulation of these systems is quite cumbersome (see, e.g., [2, 4–7]). Formalisms that would cover both aspects remain - for various reasons - anchored in the techniques and philosophy germane to one or the other of the two domains. Indeed, [5, 6] is attached to continuous techniques, while [2, 4, 7] favor a discrete event approach. Because of this, existing simulation packages and techniques that are supposed to address complex hybrid systems

[1]We note that this "definition" excludes continuous systems with chaotic behavior. Indeed, while in these systems there are no discrete events, after a certain time, the dynamics cannot be anticipated (predicted) and no "fix" - continuous or discrete - could restore this predictability.

meet with only limited success. They remain either difficult to build from scratch or inaccurate when assembled from off the shelf components; see, e.g., the problem of time step selection in EPOCHS [8]; the small timing errors intrinsic TrueTime's management of events [9]; and the long execution times of discrete event models implemented with Modelica [10].

The goal of this paper is to address the problem of managing time in simulations of Type 3 systems by proposing an approach that is truly hybrid in spirit, convenient from the engineering viewpoint, and efficient and accurate from the users' standpoint. To begin, we present a critical analysis of the two main existing approaches. As we illustrate in Sections 3 and 4, these two approaches are practicable when the systems are essentially continuous or discrete, respectively. Our third approach, presented in Section 5, becomes interesting for truly hybrid systems. Collectively, these three approaches cover the range of hybrid systems described above.

3 Continuous System Simulation Languages

There has been a concerted effort over the last several years to build discrete event simulation packages using available continuous system simulation languages (CSSLs). The Modelica user community has been particularly active in this area (see, e.g., [10–14]). These efforts have produced several working products and suggestions for language extensions that could facilitate the construction of discrete event simulation models [15].

The specific techniques that are used to construct discrete event models with CSSLs vary widely, but there are three main themes (a nice overview is given in [12]). The most common approach is to use the time or state event features of the simulation language as a basic discrete event modeling tool (see, e.g., [10, 13, 16]). The use of state events leads to an activity scanning world view and the use of time events to an event-oriented world view (see, e.g., [2]).

A second approach is used in [14] to reproduce much of the Arena discrete simulation package in the form of a library that can be used from within a Modelica model. Here, essential discrete event functionality is implemented outside of Modelica using the C language. The external function facilities of Modelica are used to access these capabilities.

Several research groups have approached hybrid system simulation by compiling Modelica models into a discrete event simulation [15, 17]. The implementation described in [17] uses quantized state integration to handle continuously evolving components, and in this way resembles the simulation tools described in Section 4. In [15], an existing Modelica compiler is extended to include new language features for discrete event system modeling.

3.1 Advantages

The most compelling advantage of this approach is its strong support for simulating complex continuous systems. This includes both advanced numerical algorithms and libraries of reusable component models. Advanced CSSLs substantially reduce simulator development time by allowing developers to work directly with the mathematical model. The compiler automates the symbolic manipulations and code production that are needed to create a working simulation. This saves time, avoids low level programming errors, and is, consequently, essential for modeling large, complex systems.

3.2 Disadvantages

The primary disadvantage of this method is rooted in the language features that make CSSLs what they are. CSSLs are designed for numerical computation. As a result, they are missing many of the features that are present in languages favored by builders of discrete event simulation software. These missing features include dynamic memory management and objects that support run-time binding of methods (i.e., objects in the sense of object oriented programming languages such as Java and C++ [18]).

This lack of general purpose, object-oriented language features seriously hampers the construction of essential data structures such as lists, queues, sets, and maps. Packet level network models and manufacturing process models are two concrete examples of discrete event systems that require these kinds of dynamic data structures. It is worth noting that many popular discrete event simulation tools, both commercial and academic - examples include OPNET, OMNEST and OMNeT++, GloMoSim, NS-2 and NS-3, and Flexsim - use general purpose programming languages to define new dynamic components.

Extended CSSLs that include strong support for object oriented programming, dynamic memory management, and other common features of general purpose programming language could greatly facilitate the construction of discrete event models. However, new language features necessarily expand the scope for programmer errors (this is particularly true of dynamic memory management), complicates compiler implementation, and generally make the language more difficult to use. The extent to which CSSLs can be extended while preserving their basic utility as a continuous system simulation tool remains to be seen.

3.3 Type 1 Example: Automotive Engine Control

A four-stroke internal combustion engine has an inherently hybrid representation. The power train and air pressure dynamics are continuous processes, whereas the pistons are modeled with four discrete operating states: intake, compression, combustion, and exhaust [19]. The time interval separating subsequent discrete states depends on the continuous motion of the power train, which in turn depends on the torque produced by the pistons.

This system is a Type 1 hybrid system. Discrete events in this model are triggered, almost exclusively, by threshold crossings of a *non-trivial* continuous function. Existing continuous system simulation languages, such as Modelica, can model these types of processes efficiently and accurately.

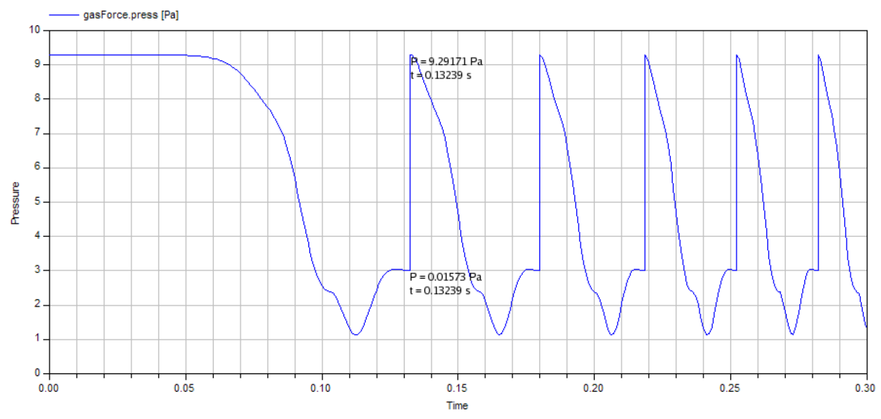


Figure 2: Pressure as a function of time in a single cylinder engine. The discrete changes in pressure are readily apparent.

The classification of this model as a Type 1 model is justified by Fig. 2, which shows the variation of air pressure with time inside of a single cylinder engine. The smooth variation in pressure between combustion events is modeled by continuous equations, but the rapid pressure changes immediately following combustion are modeled by a discrete event; that is, by an instantaneous change in pressure. We simulated the model using DYMOLA, a commercial tool that implements the Modelica language. Discontinuities in the pressure function are readily apparent. Note, for example, the instantaneous change in pressure at $t \approx 0.13$.

This combustion engine model is a fairly typical example of a system that is handled well by available continuous system modeling packages. Discrete dynamics in these types of models are characterized by events that are conditional on continuous variables satisfying logical statements. Figure 2 nicely illustrates the kind of trajectory that is characteristic of Type 1 systems.

4 Discrete Event Simulation Approaches

Combined continuous/discrete event simulation capabilities are available in most discrete event simulation tools. Two seemingly predominate approaches are discussed here. Both techniques allow for, and even promote, direct interactions between continuous and discrete variables in a hybrid model. Because of this, simulators that support these approaches must have tightly coupled continuous and discrete event simulation algorithms.

The Generalized Discrete Event System Specification (GDEVS [7, 20]) is an overarching formalization of several hybrid simulation techniques couched in terms of the Discrete Event System Specification (DEVS). DEVS is a modular modeling formalism for discrete event systems. GDEVS based approaches seek to preserve this modular modeling approach by allowing any desired decomposition of the hybrid system model. In particular, GDEVS-like techniques allow separating the event detection and continuous dynamics of a hybrid process into separate sub-components.

Within a GDEVS-like modeling framework, continuous sub-components exchange the coefficients of polynomial functions that approximate their internal dynamics, thereby simulating continuous interaction in a modular and event driven way. With this technique it is possible to approximate a coupled hybrid system as a coupled DEVS with an identical structure [2, 7, 21]. Consequently, the hybrid system can be simulated directly using discrete event simulation software.

A different approach to hybrid simulation is frequently adopted by non-modular simulation frameworks. In this approach, a numerical method is globally applied to evolve continuous variables in a combined model. Interactions between continuous and discrete event sub-components are not restricted to rigid interfaces. In an implementation, conceptually distinct sub-processes frequently access the internal state variables of other sub-processes (see, e.g., [22, 23]).

Because access to state variables is not explicitly regulated by the simulation engine, it is essential that all state variables be up to date at each simulation event time. Consequently, the integration scheme used to evolve continuous variables is evaluated whenever a discrete event occurs, as well as at time points required to control numerical errors and process events due to threshold crossings of continuous variables.

4.1 Advantages

Both of these approaches allow discrete event and continuous processes to be freely intermingled, and this gives the modeler a great deal of flexibility when building the simulator. Any decomposition of the system into sub-components can be simulated directly, and so there are no special restrictions in this respect when building hybrid models. Moreover, existing discrete event simulation software can be easily extended to support this kind of hybrid simulation approach.

4.2 Disadvantages

The tight coupling of discrete event and continuous simulation algorithms confers distinct disadvantages. When a continuous simulation algorithm is added to a non-modular simulation framework, there is a significant increase of the computational cost (in terms of time) for large simulations. When a modular framework uses a GDEVS-like approach, it is possible to introduce subtle, but significant, numerical errors that can have a dramatic effect on the simulated model behavior.

The increased computational cost of a non-modular combined continuous / discrete event simulation is a direct result of evaluating the integration scheme at each event. If the derivative function is expensive to compute, then the event execution time explodes. Frequent events combined with computationally complex continuous behaviors make the simulation grind to a halt.

This scalability problem is avoided by GDEVS-like schemes because a modular structure explicitly limits the scope of sub-component interactions. However, this is also the greatest barrier to an effective implementation: because the unrestricted decomposition of the system's model is reflected in the simulator, it may happen that the simulator does not produce an accurate calculation. A simulator for a bouncing ball gives a simple example of how this can occur.

The model of the ball has two part. The first describes how the ball moves through the air:

$$\begin{aligned} \dot{h} &= v \\ \dot{v} &= -g \end{aligned} \tag{2}$$

where h is the height of the ball, v is the velocity, and g acceleration due to gravity. The second part dictates the ball's behavior when it strikes the floor: when this event occurs, the ball rebounds, changing its velocity immediately. The condition for the event's occurrence and its consequence are

$$h = 0 \;\&\; v < 0 \implies v \leftarrow -v \tag{3}$$

where $\leftarrow$ denotes assignment.

Suppose that this model is partitioned into two processes. The first process simulates the continuous trajectory given by Eqn. 2. The output of this continuous process is a piecewise polynomial function that describes h. The second process watches this polynomial for satisfaction of Eqn. 3. When this occurs, it generates an event for the first process to change the velocity of the ball. Figure 3 illustrates this decomposition.

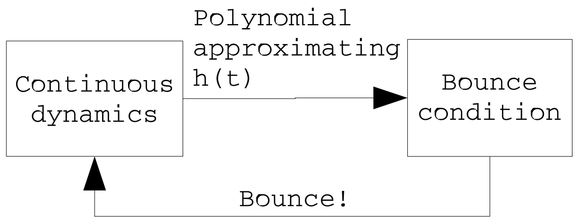


Figure 3: Decomposition of the bouncing ball into two interacting sub-processes.

If we solve Eqn. 2 with an implicit, first order accurate Euler integration scheme, then it is reasonable to use a line as the first process's output; this line is

$$h_n + v_n(t - t_n) \tag{4}$$

where h_n and v_n are the most recent values of h and v, computed at time t_n. The second process *extrapolates* using Eqn. 4 to find the point at which Eqn.

3 is satisfied. This extrapolation produces collision times that are inconsistent with $h = 0$ as computed by the first process. Consequently, the system exhibits serious simulation artifacts. Indeed, the ball appears to be bouncing on an uneven surface.

This could have been anticipated by observing that the height of the ball over a single integration time step is computed as

$$v_{n+1} = v_n - \Delta t g$$
$$h_{n+1} = h_n + \Delta t v_{n+1} = h_n + \Delta t(v_n - \Delta t g) \ .$$

The height of the ball obtained by extrapolating with Eqn. 4 is

$$\tilde{h}_{n+1} = h_n + \Delta t v_n \ .$$

The difference is

$$\tilde{h}_{n+1} - h_{n+1} = -(\Delta t)^2 g \ .$$

The effect of this discrepancy is shown in Fig. 4 for two different implementations of the model; one using the GDEVS approach and the other a method for simulating Type 1 models (a variable step integrator and a state event locator using the interval bisection method; see, e.g., [24]).

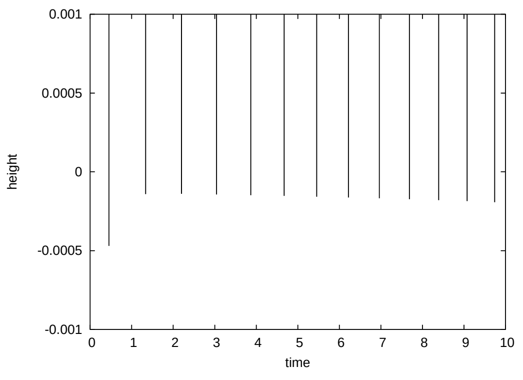
This particular problem can be alleviated by ensuring that the integration scheme and extrapolation scheme produce consistent results. For example, using an explicit Euler integration scheme would remove the event detection error (Eqn. 4 is, in fact, the explicit Euler scheme). Alternatively, we could use a second order interpolating polynomial (which, in this case, completely characterizes the dynamics of the falling ball). In general, where these types of problems emerge, their solution will require a careful, and therefore restricted, selection of algorithms for numerical integration and event detection.

4.3 Type 2 Example: Automated Manufacturing Processes

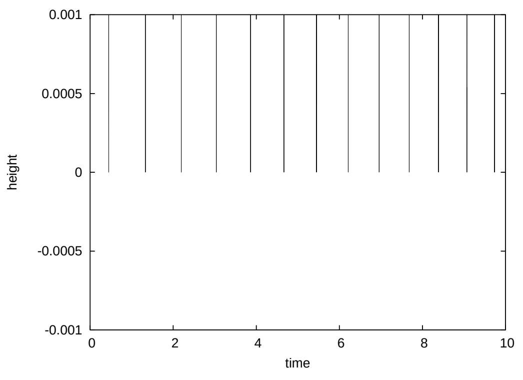
A manufacturing process model, such as that described in [25], is an excellent illustration of a Type 2 system that is handled well by existing discrete event modeling tools. This particular model characterizes the machining of a part with a physical state subject to continuous time dynamics. The queuing of raw and semi-finished materials at machining stations is modeled with discrete events.

Figure 5 shows a single manufacturing stage in this model. This stage consists of a discrete arrival process and queue. The machining tool is activated whenever it is idle and there is a part to work on. It ejects discrete parts. The machining process is modeled with a differential equation.

To demonstrate a Type 2 system, we implemented this single stage manufacturing model with Arena. The Arena discrete event simulation package is an example of a non-modular modeling and simulation tool that effectively supports modeling of Type 2 systems. Its continuous system modeling and simulation capabilities are described in [26]. Continuous trajectories are simulating using



(a) GDEVS with implicit Euler and linear extrapolation



(b) Implicit Euler with interval bisection

Figure 4: Simulation artifacts in the bouncing ball problem.

either an explicit, variable time step Runge-Kutta scheme or a simple fixed step Euler scheme.

The state event detection scheme used in Arena is relatively sophisticated. The user specifies event thresholds on continuous variables, and the simulator will search for crossings of those threshold values by retrying integration steps. Arena advances every continuous sub-component at every event time to ensure the continuous variables are up to date whenever a discrete event handler needs to access them. In this way, combined continuous and discrete dynamics are accurately simulated.

For this example, the queue capacity is infinite and blanks arrive at intervals determined by a uniform random variable with a range of $[0.1, 10]$. The machining process is described by

$$\dot{x} = k(1.1 - x)$$

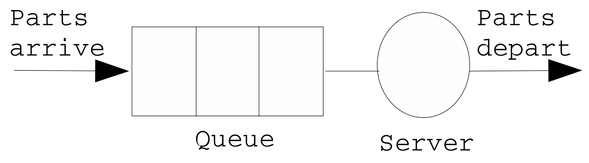


Figure 5: Model of a single manufacturing stage.

where k is a discrete variable that turns the machine on when a part arrives and off when a part is completed. The machining process starts with $x = 0$ and finishes when $x = 1$.

This manufacturing model is illustrative of systems that are easily modeled with existing discrete event modeling packages. Figure 6 shows the relative simplicity of the continuous dynamics that are generally associated with Type 2 models.

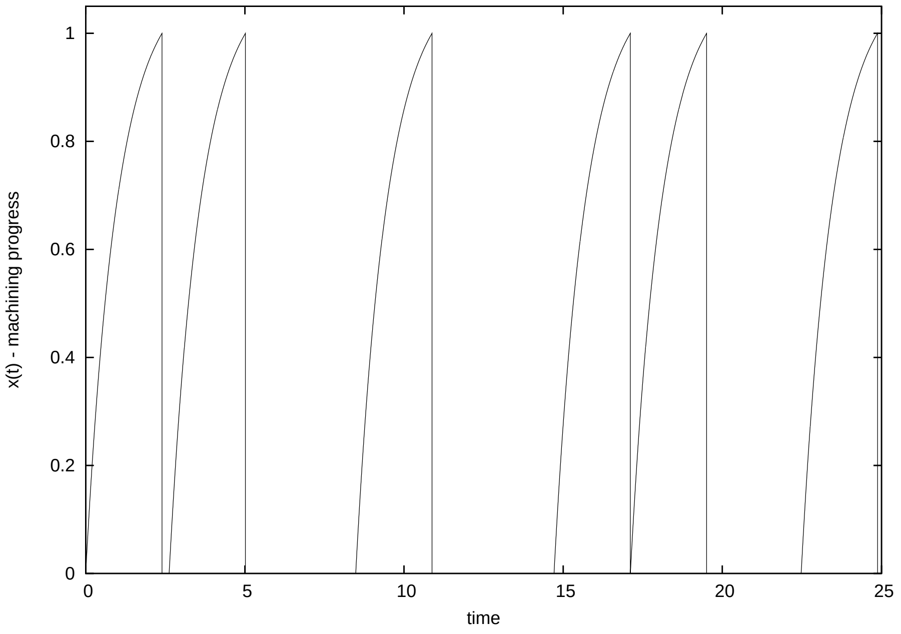


Figure 6: The continuous machining process in one simulation run of the single stage manufacturing process.

5 Split Hybrid System Modeling

The split hybrid system modeling approach explicitly recognizes discrete event and continuous variables in a system model, and this knowledge is used to construct an efficient simulator. We simulate individual sub-components using the most appropriate algorithms: numerical integration methods for continuous components and discrete event algorithms for discrete components. This overcomes the major limitation of CSSL and discrete event simulation techniques, where the simulation approach assumes the predominance of a particular type of component.

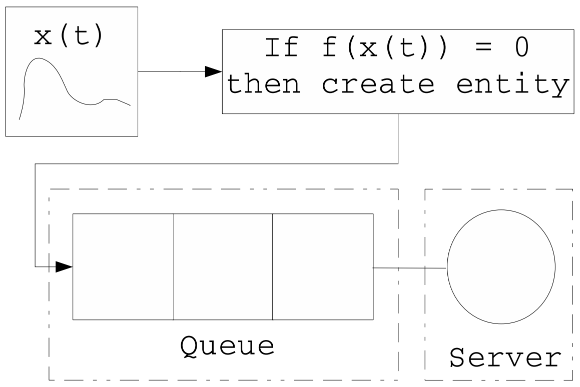
The idea underlying our approach is simple. The overarching modeling paradigm is based on discrete events. Continuously interacting sub-components are treated as a single entity. The internal dynamics of these entities are simulated using any suitable numerical method. However, interactions with other components occur at time and state events through an explicitly defined interface. These events are the only mechanism for interacting with the continuous entities.

This strict and explicit disentanglement of discrete and continuous dynamics distinguishes the split hybrid modeling approach from its predecessors. By focusing on the (large) subclass of split hybrid systems, we avoid the numerical problems inherent in GDEVS. The computational problems of a non-modular modeling approach are overcome by restricting discrete/continuous interactions to specific discrete events. This clear distinction between continuous and discrete components allows us to use sophisticated continuous simulation algorithms within the confines of a general purpose discrete event simulation framework.

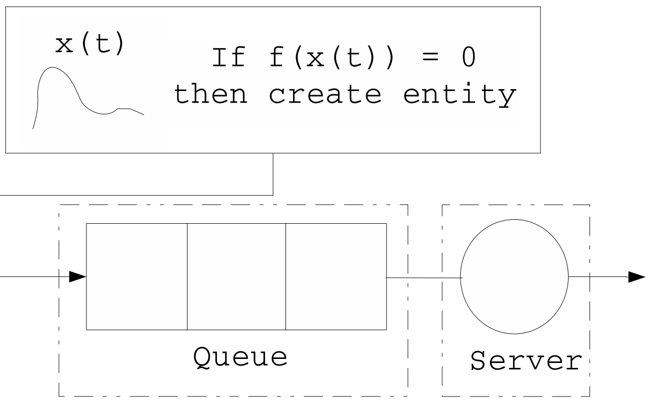
Figure 7 illustrates this disentanglement principle by comparing two decompositions of a hybrid model. This model has four components, two of which interact continuously. Figure 7(a) shows a decomposition that is disallowed in our approach, but permitted by GDEVS. A GDEVS-like approach allows the continuously interacting components to be described and simulated as separate blocks. Figure 7(b) reorganizes this system to be compatible with the split hybrid system modeling approach.

This modular, self-contained description of the continuous sub-components explicitly identifies and distinguishes continuous and discrete event dynamics. The simulation algorithm takes advantage of this by using continuous system simulation algorithms to generate the internal dynamics of continuous blocks. Interactions with other discrete event components is coordinated through the discrete event interface of the continuous model. In this way, proper coordination of the discrete and continuous components is assured, accurate continuous trajectories are generated, and the resulting simulation software is efficiently executed.

The split hybrid system modeling approach describes self-contained, continuous sub-components with four functions. The evolution function F describes how the continuous, internal state variables evolve between discrete events. The event scheduling function G indicates how much time will elapse before the next discrete internal event. The discrete action function A describes how the dis-



(a) Incompatible decomposition



(b) Compatible decomposition

Figure 7: Two decompositions of a hybrid system, one compatible and the other incompatible with the split hybrid system approach. Solid blocks interact continuously with each other; dotted blocks are purely discrete.

crete state changes in response to internal (state) and external (input) events. The discrete output function L describes how the output changes in conjunction with discrete internal events.

Continuous sub-components are formalized with a structure

$$\begin{aligned}
&X \text{ and } Y \text{ , the input and output value sets}\\
&S \text{ , the internal state set}\\
&F : S \times \mathbb{R} \rightarrow S \text{ , the evolution function}\\
&G : S \rightarrow \mathbb{R} \text{ , the event scheduling function}\\
&A : S \times X_\Phi \rightarrow S \text{ , the discrete action function,}\\
&\text{where } X_\Phi = X \cup \{\Phi\} \text{ and } \Phi \text{ is the non-event, and}\\
&L : S \rightarrow Y \text{ , the discrete output function.}
\end{aligned} \tag{5}$$

The set X is the range of values that can be injected into the system. The set Y is the range of values that can be produced by the system. The set S is the range of the system's internal state variables.

The evolution function $F(q, h)$, with $q \in S$ and $h \in \mathbb{R}$, takes the system from a state q at time t to a later state q' at time $t + h$. This function describes continuous, autonomous evolution of the model's internal state. The system evolves continuously until $G(q) = 0$ or a change occurs in the input trajectory. At these points, the discrete action function dictates an immediate and, possibly, discontinuous state change.

The discrete action function $A(q, u)$, with $u \in X_\Phi$, determines the response of the model to discrete events. These events can be inputs to the system or be triggered by its internal dynamics. In either case, the system changes state instantaneously from q to $q' = A(q, u)$. Changes due to internal dynamics occur when $G(q) = 0$, and the subsequent state of the system is determined by $A(q, \Phi)$. External events are due to a change in the input trajectory. In this case, the state immediately following the event is given by $A(q, x)$, where $x \in X$ is the value of the input trajectory immediately after the input event occurs.

The discrete output function $L(q)$ defines the model's output trajectory. The initial output value is given by $L(q_0)$, where q_0 is the initial state of the system. Discrete changes in the output trajectory occur when $G(q) = 0$. At these times, the output trajectory takes the value $L(q)$, and it keeps this value until the system again enters a state in which G evaluates to zero.

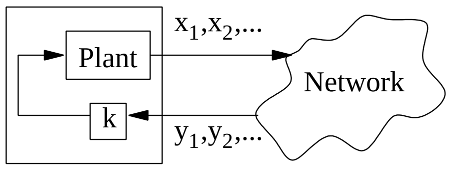


Figure 8: Illustrative model of proportional gain control through a network. The subscripted x are samples of the plant's continuous state variable that are sent through the network; the subscripted y are the same samples received by the controller through the network.

A simple example will illustrate how these functions are used to define a hybrid model. Consider the proportional gain controller illustrated in Figure 8. The continuous state variable x of the plant is sampled at intervals Δt and these samples travel through a shared network (e.g., an Ethernet) that introduces some delay before they are multiplied by the control gain k and fed back into the plant. The continuous elements of this model comprise three state variables: the plant state x, the output sample y last received from the network, and the time t_e that has elapsed since a sample of x was last obtained. Assume that the differential equation governing the plant between changes in y is

$$\dot{x} = x - ky \tag{6}$$

If explicit Euler is used to solve this equation, then its simulation model in terms of the structure 5 is

$$\begin{aligned}
&X = Y = \mathbb{R} \\
&S = \{(x, y, t_e) \mid x, y \in \mathbb{R} \ \& \ 0 \leq t_e \leq \Delta t\} \\
&F((x, y, t_e), h) = (x + h(x - ky), y, t_e + h) \\
&G((x, y, t_e)) = \Delta t - t_e \\
&A((x, y, t_e), u) = \begin{cases} (x, y, 0) & \text{if } u = \Phi \\ (x, u, t_e) & \text{otherwise} \end{cases} \\
&L((x, y, t_e)) = x
\end{aligned}$$

The dynamics associated with the structure 5 can be described in three ways: as a Discrete Event System Specification (DEVS) atomic model (see, e.g., [2]), a Hybrid Input/Output Automata (HIOA) (see, e.g., [5]), and algorithmically as an event scheduling simulation. In what follows, we detail these three equivalent descriptions. In doing so, the class of split hybrid systems is described in the context of well established analysis and simulation frameworks for discrete event and hybrid systems.

5.1 DEVS

Our first characterization associates the structure (5) with a DEVS atomic model. The atomic model's set of states is S, and its input and output sets are X and Y. The state transition, time advance, and output function are defined in terms of the evolution function F, event scheduling function G, discrete output function L, and discrete action function A. These definitions are

$$\begin{aligned}
&\delta_{int}(q) = A(F(q, ta(q)), \Phi) \\
&\delta_{ext}(q, e, x) = A(F(q, e), x) \\
&\delta_{con}(q, x) = A(F(q, ta(q)), x) \\
&ta(q) = G(q) \\
&\lambda(q) = L(F(q, ta(q))) \ .
\end{aligned} \tag{7}$$

Output and internal events coincide with the expiration of the time advance. The discrete output is computed using the system state just prior to the discrete event (i.e., prior to applying the discrete action function).

An implementation of this atomic model will, in general, require events that do not result in discrete actions (i.e., an evaluation of A) or discrete output (i.e., an evaluation of L). These types of events are needed, for instance, when the evolution and event scheduling functions are implemented with numerical integration and state event detection algorithms (see, e.g., [24]). A DEVS model that is functionally equivalent to (7) can be had by defining a function

$$IntegStep : S \to \mathbb{R}$$

that picks the next integration step size. The system dynamics are then defined by

$$\begin{aligned}
&\delta_{int}(q) = \\
&\quad \begin{cases} A(F(q, ta(q)), \Phi) & \text{if } G(q) \leq \mathit{IntegStep}(q) \\ F(q, ta(q)) & \text{otherwise} \end{cases} \\
&\delta_{ext}(q, e, x) = A(F(q, e), x) \\
&\delta_{con}(q, x) = A(F(q, ta(q)), x) \\
&ta(q) = \min\{G(q), \mathit{IntegStep}(q)\} \\
&\lambda(q) = \\
&\quad \begin{cases} L(F(q, ta(q))) & \text{if } G(q) \leq \mathit{IntegStep}(q) \\ \Phi & \text{otherwise .} \end{cases}
\end{aligned} \tag{8}$$

5.2 HIOA

We can also characterize the structure (5) by associating it with an appropriate subclass of HIOA. Let $x(t)$, $y(t)$, and $q(t)$ denote points on the input, output, and state trajectories of the HIOA. The notation $x(t^-)$ is used to refer to the value of the function x when t is approached from the left, and $x(t^+)$ the value when t is approached from the right. Any particular HIOA is acceptable so long as it satisfies the following restrictions.

1. A set of internal variables is defined whose range is the set of states S.
2. A set of input variables is defined whose range is the set of inputs X.
3. A set of output variables is defined whose range is the set of outputs Y.
4. Input and output variables are discrete (i.e., their trajectories are piecewise constant functions).
5. $y(t) = L(q(t^*))$ where

$$\begin{aligned}
&t^* \leq t \;\& \\
&(G(q(t^*)) = 0 \text{ or } t^* = 0) \;\& \\
&\forall \tau \in (t^*, t], G(q(\tau)) \neq 0.
\end{aligned}$$

   .

6. If $G(q(t)) = 0$ and $x(t^-) = x(t^+)$, then the subsequent internal state $q'(t)$ is given by $q'(t) = A(q(t), \Phi)$.
7. If $x(t^-) \neq x(t^+)$, then the subsequent internal state $q'(t)$ is given by $q'(t) = A(q(t), x(t^+))$.
8. Otherwise, the trajectory $q(t)$ is dictated by the evolution function F.

Items 5, 6, 7, and 8 impose a particular form on the system trajectories. These conditions are sufficient for the hybrid automaton to have a DEVS representation in the form of (7). The advantage of this is that the hybrid automaton can be simulated with an event scheduling algorithm, and therefore can be integrated directly with a discrete event simulation tool.

Condition 5 forces change in the automaton output variables to coincide with internal states that cause the function G to be zero. The output at this time is dictated by the discrete output function L, and the output value is maintained until the next state for which G is zero.

Conditions 6, 7, and 8 allow for explicit scheduling of discrete changes to the internal state of the hybrid automaton. This preserves the semantics of the DEVS time advance function, and makes possible a discrete event simulation of the system.

5.3 Event Scheduling

Lastly, we describe the dynamic behavior of structure (5) in terms of an event oriented simulation program. For this purpose, the hybrid system is assumed to be contained within a logical process, or to be otherwise partitioned from the rest of the discrete event model (see, e.g., [27]). Three types of events are required. A *Step* event performs an integration step in which discrete actions do not occur. A *Change* event performs an integration step at the end of which a discrete action does occur. An *External(x)* event describes a change in the discrete input to the system. The discrete input value is denoted by x.

Let q denote the state of the logical process, h the preferred time step for integration scheme, Φ denote an absence of input events, t_l be the last event time, and t be the current time. Algorithms 1, 2, and 3 describe the processing required at *Step*, *Change*, and *External(x)* events.

Algorithm 1 *Step* event

```
Cancel pending Step and Change events
$q' \leftarrow F(q, t - t_l)$ {Update the continuous state variables}
$t_l \leftarrow t$
Find the next integration time step
if $t + h < t + G(q')$ then
  schedule a Step event at time $t + h$
else
  schedule a Change event at time $t + G(q')$
end if
$q \leftarrow q'$ {Store the new variable values}
```

When *External(x)* and *Change* events coincide, it is preferable to define a fourth event type to handle this case (see [28–30] for a discussion of some issues surrounding this fourth event type). In the DEVS formalization, this fourth event is described by the confluent transition function. It is implicit in restrictions 5, 6, 7, and 8 of the HIOA formalization.

Algorithm 2 *Change* event

```
Cancel pending Step and Change events
$q' \leftarrow F(q, t - t_l)$ {Update the continuous state variables}
Schedule output events at $t$ using the output value $L(q')$
$q'' \leftarrow A(q', \Phi)$ {Apply the discrete event}
$t_l \leftarrow t$
Find the next integration time step
if $t + h < t + G(q'')$ then
  schedule a Step event at time $t + h$
else
  schedule a Change event at time $t + G(q'')$
end if
$q \leftarrow q''$ {Store the new variable values}
```

Algorithm 3 *External(x)* event

```
Cancel pending Step and Change events
$q' \leftarrow F(q, t - t_l)$ {Update the continuous state variables}
$q'' \leftarrow A(q', x)$ {Apply the discrete event}
$t_l \leftarrow t$
Find the next integration time step
if $t + h < t + G(q'')$ then
  schedule a Step event at time $t + h$
else
  schedule a Change event at time $t + G(q'')$
end if
$q \leftarrow q''$ {Store the new variable values}
```

If this fourth event type can be defined within the simulation environment, then the desired computational steps are given as Algorithm 4. The *Confluent(x)* event allows output events to be produced using the state just before the discrete state change occurs. A subsequent *Change* event can be scheduled via the G function if additional output is desired immediately following the discrete change.

If a fourth event type is not possible, then a preferred priority for *Change* and *External(x)* events must be given based on knowledge of the system being simulated. Also note that the DEVS and HIOA definitions require that simultaneous input events be presented simultaneously to the continuous subcomponent. If this option is not available in a particular simulation tool, then some preferred prioritization of *External(x)* events must be specified as well.

5.4 Advantages

The split hybrid modeling method combines the use a general purpose discrete event simulator with advanced numerical algorithms for handling continuous

Algorithm 4 *Confluent(x)* event

```
Cancel any pending Step or Change events
$q' \leftarrow F(q, t - t_l)$ {Update the continuous state variables}
Schedule output events at $t$ using the output value $L(q')$
$q'' \leftarrow A(q', x)$ {Apply the discrete event}
$t_l \leftarrow t$
Find the next integration time step
if $t + h < t + G(q'')$ then
  schedule a Step event at time $t + h$
else
  schedule a Change event at time $t + G(q'')$
end if
$q \leftarrow q''$ {Store the new variable values}
```

dynamics. Continuous blocks can be implemented directly or generated automatically by CSSL compilers that produce software modules with a suitable programming interface (e.g., the acslXtreme API produced by the AEgis ACSL compiler). By combining advanced discrete event simulation tools and state of the art continuous modeling capabilities, it is possible to build large and complex hybrid system simulations that are numerically robust and computationally efficient.

The three main disadvantages of CSSL, GDEVS-like, and non-modular combined simulations are overcome by the split hybrid modeling approach. The overarching discrete event modeling and simulation framework ensures an adequate set of basic structures for describing discrete event dynamics. The computationally tractability problem of a non-modular approach is eliminated by restricting discrete/continuous interactions to explicitly defined input and output events.

By forcing continuously interacting elements into atomic blocks, our approach allows the numerical integration and state event detection schemes to be mutually consistent *without restricting the choice of algorithms*. The use of consistent schemes eliminates the simulation artifacts that can emerge in a GDEVS-like approach. At the same time, the simulation builder can use any desired set of numerical techniques to simulate continuous processes.

For instance, consider the bouncing ball problem discussed in Section 4.2. The GDEVS solution gets into trouble because it uses two different approximations of the ball's trajectory, namely (i) an implicit Euler approximation to the differential equations and (ii) a linear extrapolation for anticipating bounce events. The GDEVS solution requires these two different approximations because the event detector cannot access the state of the integrator and vice versa. The solution shown in Fig. 4(a) is the result of this inconsistency.

The two parts of the ball's dynamics interact continuously because the event condition depends on the continuously evolving height of the ball. Consequently, the split system approach requires that they be combined into a single modeling entity. Combining the event condition and continuous dynamics makes it possible to use a coordinated state event detector and numerical solver. In particular, we can use the interval bisection method to find bounce events and produce the desired solution shown in Fig. 4(b) (see, e.g., [24]).

For simple models it is often possible to construct an adequate approximation from an almost intuitive understanding of the dynamics. The bouncing ball is an extreme case where the solution of the corresponding differential equation is easily written down and bounce events can be trivially anticipated. Complex hybrid dynamics, on the other hand, have non-intuitive continuous dynamics and events that are difficult to anticipate. Satisfactory simulation of these models requires the use of the split approach.

5.5 Disadvantages

The success of our proposed approach is predicated on a suitable split of the hybrid system into discrete and continuous dynamics.[2] This is avoided by the previous approaches (i.e., Section 4 and Section 3), which allow for direction simulation of any model decomposition. There are at least two dangers in requiring an explicit decomposition. The first is that such a description will be so unnatural and cumbersome that it, by itself, defeats the modeling effort. This, at least, can be determined early and another (less accurate, precise, or computationally efficient) approach attempted.

The second danger is more subtle, occurring when the model *description* is not split but the model *implementation* is split by the programmer. In practice, this after the fact decomposition can create two different software artifacts: the model description and its simulation software. This substantially increases the possibility of errors in both the paper model description and its computer implementation (this is a well-known software engineering problem [18]). This danger is also avoided by the previous approaches, which allow any model decomposition to be translated directly into a software implementation.

The greatest strength of the proposed approach is reuse of existing continuous system simulation tools within a discrete event simulation framework. Unfortunately, most current continuous system simulation packages support only half of the features needed to do this. The two features that are widely supported are

1. an interface for setting and getting the values of continuous variables and
2. functions or methods for evaluating a single step of the integrator.

The missing features are, however, crucial for applying the split hybrid system modeling approach. These features are

3. functions or methods for getting the next step selected by the numerical solver *without* actually committing to that step and

[2]This assumes that a useful decomposition can be found for the particular system of interest.

4. separation of the numerical integration step and state event evaluation.

To summarize, the shortcoming of existing continuous system simulation software is that the step size selection, integration step, and application of state events are, from the end user perspective, rolled into a single function or method. These three activities need to be split into three distinct, end user controlled phases to allow proper event scheduling, output event production, and input event handling (i.e., as per the formulations in Sections 5.1, 5.2, and 5.3).

5.6 Type 3 Example: An Automatic Load Control System

The efficient and reliable delivery of electric power increasingly depends on networked SCADA (Supervisory Control and Data Acquisition) and distributed control systems. These systems often operate over commercially available, frequently IP based, communication networks. Problems of control and communication in the smart electric grid have recently focused attention on modeling and simulation of distributed, wide area control systems in this context (see, e.g., [8, 31]).

A system for under frequency protection of wide area electric power systems illustrates several aspects of Type 3 hybrid systems. The objective of this system is to prevent under frequency generator failures by making small and rapid changes to the network load. The idea is to incur small service interruptions when under frequency failures are imminent, and then to automatically restore service when the system stabilizes.

5.6.1 Continuous elements

The continuous component in this system is a power generation and transmission model of a 17 bus system that is derived from the IEEE 14 bus system. The model consists of twelve loads and five generators that are interconnected as shown in Fig. 9. Frequency, and not voltage, disturbances are the focus of the current investigation. Therefore, only real power flow is considered [32].

The generators are modeled as synchronous machines using the swing equation plus additional equations that model a governor, non-reheat turbine, and over speed breaker. One of the five generators also includes a basic Automatic Generation Control (AGC) unit that eliminates steady-state frequency error throughout the system. The three equations that describe the generator dynamics are

$$\begin{aligned}
\Delta\dot{\delta}_g &= \Delta\omega \\
\Delta\dot{\omega} &= (\Delta P_m - \Delta P_e)/M \\
\Delta\ddot{P}_m &= -100(k_{agc}\Delta\delta_g + \Delta\omega/R + 0.25\Delta\dot{P}_m + P_m)
\end{aligned}$$

where ΔP_m and ΔP_e are the deviations of mechanical power output and electrical demand from the initial steady state operating point, $\Delta\delta_g$ is the change in relative generator shaft angle, and $\Delta\omega$ is the deviation of the shaft angular velocity from 60 Hz. The values R, M, and k_{agc} are the generator's speed

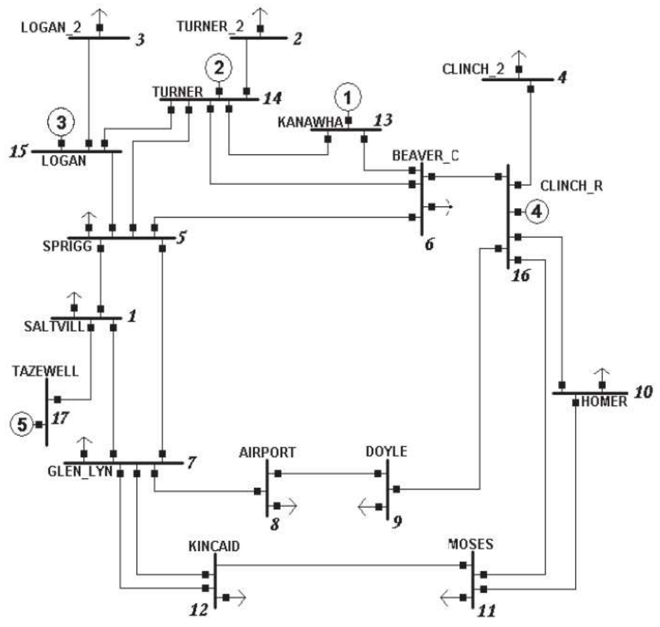


Figure 9: The 5 generator and 12 load bus power system model.

droop constant, rotational inertia, and the AGC gain. If the speed deviation of a machine exceeds ±0.1 Hz, then it is disconnected from the transmission network.

Real power flow is calculated using known generator shaft angles and electrical power demand at the load buses. Disconnected generators are treated as load buses with zero power demand [32]. To facilitate the calculation of electrical demand P_e on the generators, the network admittance matrix Y is broken into the four sub-blocks shown in Eqn. 9. The Y_{ll} block describes load to load connections, Y_{lg} and Y_{gl} describe the symmetric generator-load/load-generator connections, and Y_{gg} describes generator to generator connections. Similarly, the bus angle and electric power vectors are split into upper and lower blocks. The vectors $\bar{\Theta}_l$ and $\bar{P}_l$ denote the load bus angles and injected power. The vectors $\bar{P}_e$ and $\bar{\Theta}_g$ are the electrical demand on the generators and the generator shaft angles. The power flow equations are

$$\begin{bmatrix}\bar{P}_l \\ \bar{P}_e\end{bmatrix} = \begin{bmatrix}Y_{ll} & Y_{lg} \\ Y_{gl} & Y_{gg}\end{bmatrix}\begin{bmatrix}\bar{\Theta}_l \\ \bar{\Theta}_g\end{bmatrix} \ . \tag{9}$$

The electrical demand on the generators is given by

$$\begin{aligned}\bar{\Theta}_l &= Y_{ll}^{-1}(\bar{P}_l - Y_{lg}\bar{\Theta}_g) \\ \bar{P}_e &= Y_{gl}\bar{\Theta}_l + Y_{gg}\bar{\Theta}_g \ .\end{aligned}$$

Attached to each generator is a monitor that is frequency sensitive. The monitor is triggered when the generator frequency differs by 0.001 Hz with respect to the last triggering event, i.e., the monitor samples the generator when its frequency reaches 60.0 Hz plus and minus 0.001 Hz, 0.002 Hz, etc. and at these times it measures five quantities: the generator's mechanical power ΔP_m, rate of change in mechanical power $\Delta \dot{P}_m$, electrical load ΔP_e, shaft velocity $\Delta\omega$, and shaft acceleration $\Delta\dot{\omega}$.

Using the DEVS approach described in section 5.1, this continuous model is encapsulated in a single atomic component. Input to the component are discrete changes in the electric load; these inputs cause instantaneous change to the diagonal elements of Y_{ll}. Output from the component are sensor measurements, which occur at discrete values of the $\Delta\omega$s. The evolution function F gives the solution to the continuous generator equations at the instants of discrete input and output and at time points selected to control numerical errors. The evolution function is implemented with a fourth/fifth order Runge-Kutta (RK45) integrator with error control (see, e.g, [33]). This numerical integrator is particularly attractive because its step size can be adjusted at will to accommodate discrete events. The Template Numerical Library is used to solve the power flow equations at each integration step.

The event scheduling function G gives the smaller of i) the time remaining to the next sampling instant, ii) the next opening of a frequency protection breaker, and iii) the step size h selected by the numerical integrator. The last of these is calculated first, and this is the largest value that G will report. Items (i) and (ii) are calculated by looking for the first instant in the interval $[0, h]$ at which any of the $\Delta\omega$s reach a threshold value. This is a root finding problem and we solve it here using a relatively simple interval bisection approach (see, e.g., [24]). If no such instant exists, then G simply returns h.

The discrete action function A opens frequency protection breakers, sets the elements of Y_{ll} to indicate changes in load, or both as is required by the events that triggered the evaluation of A. In the DEVS implementation of this model, load changes are due to an external event (i.e., δ_{ext}), frequency protection breakers are opened due to an internal event (i.e., δ_{int}) that has one of the $\Delta\omega$s at the tripping threshold of 0.1 Hz, and a confluent event (i.e., δ_{con}) may cause both. The output function L returns a sample for the generators that are at a sampling threshold, or the non-event Φ if there is no such generator. *Note that the input and output trajectories for this model of the generators, loads, and transmission network comprise only discrete events.* This is the characteristic feature of the split system method.

5.6.2 Discrete elements

Samples output by the model of the generators, loads, and transmission network are input to the model of the control and communication system. At each sampling instant, the monitors at the sampled generators estimate the time to an under-frequency failure by

$$t_f = \begin{cases} \frac{60.0(\Delta\omega+1)-59.9}{|\Delta\dot{\omega}|} & \text{if } \Delta\dot{\omega} < 0.0 \\ \infty & \text{if } \Delta\dot{\omega} \geq 0.0 \end{cases}$$

and time to meet demand by

$$t_s = \frac{|\Delta P_m - \Delta P_e|}{|\Delta \dot{P}_m|} \ .$$

The generator is in danger of being disconnected if

$$t_f \leq t_s \ .$$

In this case, the monitor broadcasts a request asking all load buses to reduce their power demand. If, on the other hand,

$$t_f > kt_s$$

where $k >> 1$ is a safety factor, the system is operating under capacity. In this case, the monitor broadcasts a message indicating that demand for power can be increased.

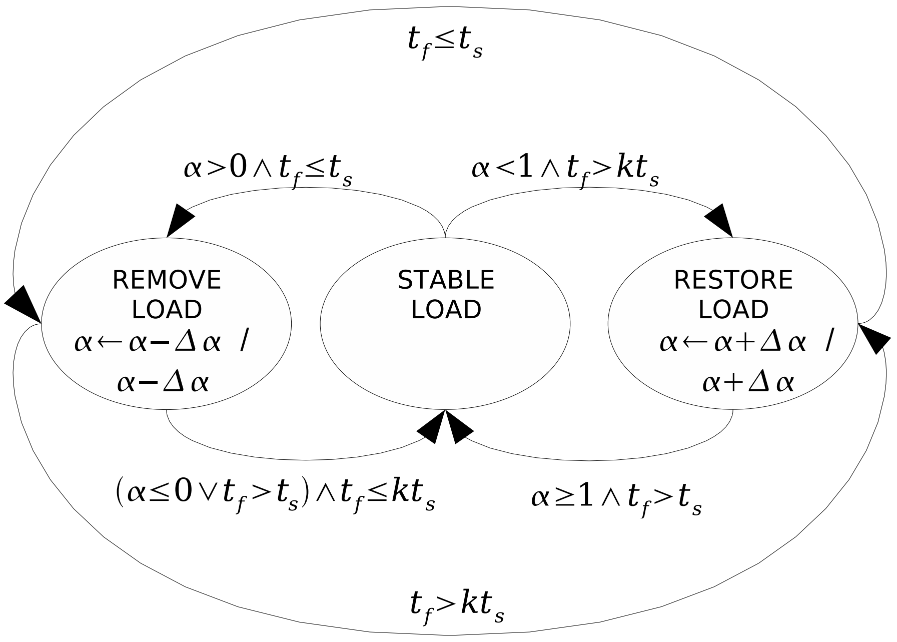
Load change requests are summarized by the load service fraction α, with $\alpha \in [0, 1]$. When $\alpha = 1$, the generator can tolerate the full electrical demand seen at its terminal. When $\alpha < 1$, the generator would like to see the demand on its terminal reduced to a fraction α of the full power demand. Changes to α occur in discrete increments $\Delta\alpha$.

The operation of the monitor is depicted in Fig. 10. The monitor state and output are computed at each sampling event. Circles denote discrete phases, and the action performed in each phase is denoted by *state change / output.* Labeled arrows denote phase change conditions. At each sampling instant, the phase change conditions are evaluated and the phase is changed accordingly. Then the output value is produced and, subsequently, the state variable change is applied. A new monitor state and output is calculated every time the monitor takes a measurement.

Load buses remember the last load service fraction received from each monitor. The remembered requests are denoted α_i, with $i \in [1, 5]$ indicating the monitor that produced the request. Each load bus is also aware of its electrical demand. On receiving a message, the load bus removes or restores some of its power demand from the transmission network. The serviced load L_s at each bus is a fraction of the total demand L_d given by

$$L_s = L_d \left(\frac{1}{5} \sum_{i=1}^{5} \alpha_i \right) \ .$$

With five generators in the system, all of the demand is serviced if the α_i are all 1, and no demand is serviced if the α_i are all zero. Note that the monitor



$t_f \le t_s$
$\alpha > 0 \wedge t_f \le t_s$
$\alpha < 1 \wedge t_f > kt_s$
REMOVE LOAD
$\alpha \leftarrow \alpha - \Delta\alpha$ / $\alpha - \Delta\alpha$
STABLE LOAD
RESTORE LOAD
$\alpha \leftarrow \alpha + \Delta\alpha$ / $\alpha + \Delta\alpha$
$(\alpha \le 0 \vee t_f > t_s) \wedge t_f \le kt_s$
$\alpha \ge 1 \wedge t_f > t_s$
$t_f > kt_s$

Figure 10: State transition diagram for the generator monitor.

attached to a disconnected generator will continue to operate with $\Delta P_e = 0$, and this causes the service fraction for the monitor to eventually settle at 1.

The monitors and load buses communicate through a packet switching network. Communication lines follow the network transmission lines, and packets are routed from origin to destination through this shared communication medium. The communication lines are modeled as queues with a fixed throughput, measured in bits per second (bps), and base delay. The time required for a packet to traverse a single line is given by

$$\frac{\text{bits}}{\text{throughput}} + \text{base delay}\ .$$

Each line has a buffer for queuing packets, and only one packet can traverse the communication line at any time. No packets are dropped. In general, a message will need to travel through several lines before reaching its destination. Network flooding is used to implement the broadcast function (see, e.g., [34]).

The control and communication system is implemented in three parts. The monitors and actuators at the loads are DEVS atomic models; these are pure discrete event components and their implementation is straightforward. The communication network is modeled using NS2, and it is encapsulated in a component whose input and output are the commands produced by the monitors and consumed by the load actuators. All of the components and their interactions are illustrated in Figure 11.

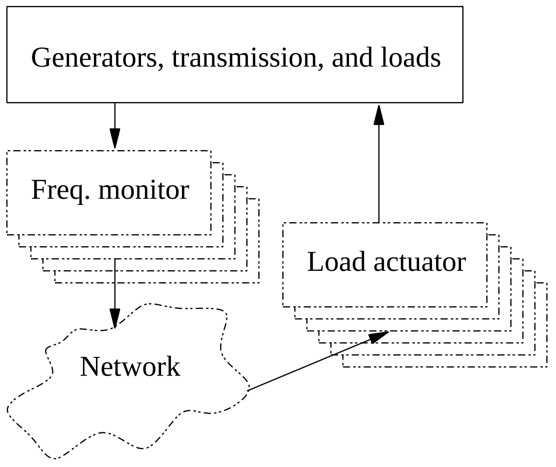


Figure 11: Components and their interconnections in the power system model. A solid outline indicates the component has continuous dynamics; a dashed outline indicates a component that is entirely discrete.

5.6.3 An experiment

One experiment will serve to demonstrate the hybrid trajectories produced by this system. The line admittances used in this experiment are described in [32]. The initial power at each generator is calculated to ensure steady state at a selected bus angle [32]. Other generator parameters are listed in Table 1. The controller parameters are $k = 10^4$ and $\Delta\alpha = 0.1$. The size of a control message is 900 bytes. The base link latency is 10 ms and throughput is 560 kbps.

Generator	$1/R$	k_{agc}
1	300	0
2	225	200
3	300	0
4	300	0
5	225	0

Table 1: Generator parameters

Table 2 shows the electrical demand schedule that is used in this experiment. The $t = 0$ column shows initial power demand at each bus. Subsequent columns contain an entry only for buses at which the power demand changes. Electrical demand is described by ‘per unit’ power injected at the load bus. Without any load control, this schedule causes all five generators to trip offline following the load spike at $t = 10$ seconds. The failure scenario is shown in Figure 12.

Load bus	$t = 0.0$	$t = 1.0$	$t = 10.0$
1	0.0		
2	-0.217		
3	-0.942		
4	-0.112		
5	-0.478		
6	-0.076		
7	-0.295	0.0	-0.4
8	-0.09	0.0	-0.09
9	-0.035		
10	-0.061	0.0	-0.4
11	-0.135	0.0	-0.135
12	-0.149	0.0	-0.149

Table 2: Electrical demand schedule

Figure 13 shows that, in this scenario, the control scheme prevents a system collapse. The frequency dips, but this is detected by the generator monitors and appropriate control messages are acted on by the load actuators. Figure 14(b) shows the total load fraction requested by the controllers. These messages are able to traverse the network rapidly enough that the load manipulations are effective. As Fig. 14(a) shows, however, the control action is delayed and very ragged due to communication delays. The combined effect on overall system behavior would be difficult to anticipate without an integrated, dynamic model of the power, control, and communication systems.

6 Conclusions

This paper proposes a split modeling approach for simulating hybrid systems. The overarching modeling paradigm is based on discrete events. Continuously interacting sub-components are treated as a single entity. These entities interact with other components through a discrete event interface that is defined in terms of time and state events. Their internal dynamics are simulated using any suitable numerical method. However, interactions with other (continuous or discrete) autonomous components occurs through well defined events. These events are the only mechanism for interacting with the continuous entities.

The split hybrid system modeling approach explicitly recognizes discrete event and continuous variables in a system model, and this knowledge is used to construct an efficient simulator. Individual sub-components are simulated using the most appropriate algorithms: numerical integration methods for continuous components and discrete event algorithms for discrete components. This intrinsic capability overcomes the major limitations of CSSL and GDEVS techniques and enables the reuse of powerful, existing continuous system simulation algorithms as part of existing discrete event simulation models. Thus, the resulting

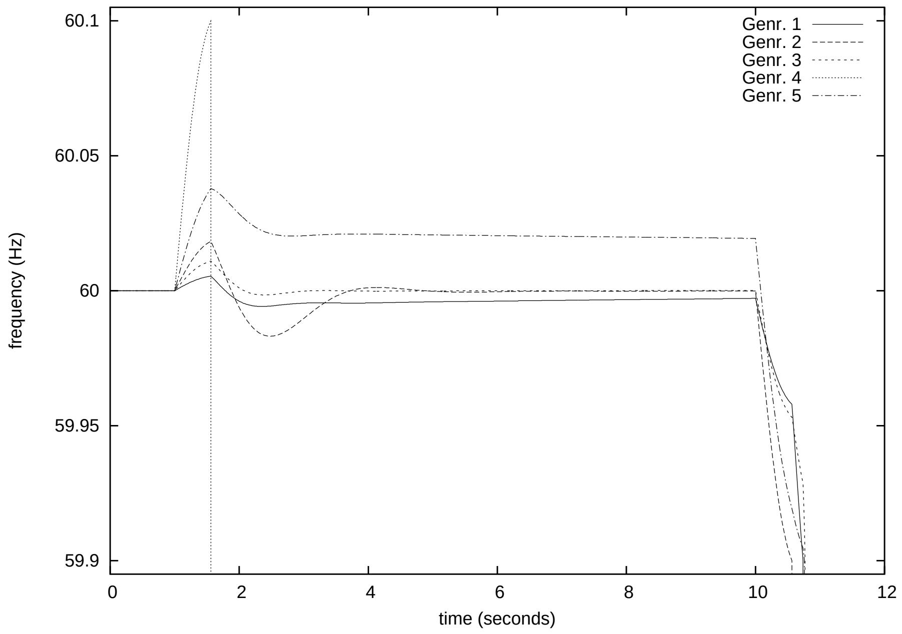


Figure 12: System failure in the absence of any load control scheme.

software is computationally efficient, ensures accurate interactions between discrete event and continuous components, and requires only a modest software integration effort.

Acknowledgments

We thank Sara Mullen and Laurie Miller at the University of Minnesota for their assistance with the power grid example used in this paper. Thanks also to Conrad Housand at AEgis Technologies for supporting our research effort with an acslXtreme license.

References

[1] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[2] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation, 2nd Edition.* Academic Press, 2000.

[3] A. Wayne Wymore. *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design.* CRC Press, Boca Raton, Florida, 1993.

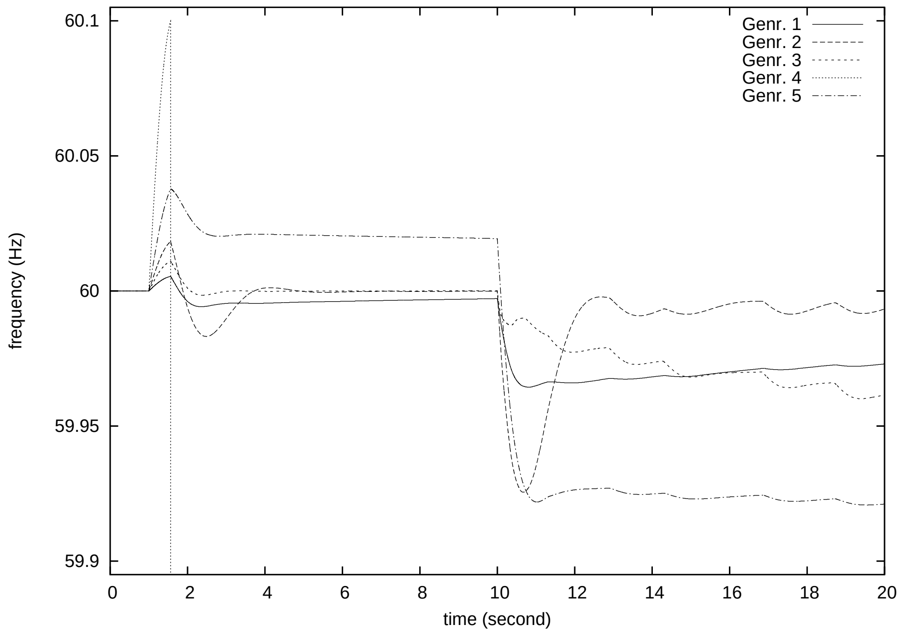


Figure 13: The load control scheme prevents system failure.

[4] Fernando Barros. Modeling and Simulation of Dynamic Structure Heterogeneous Flow Systems. *SIMULATION*, 78(1):28–35, 2002.

[5] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata. *Information and Computation*, 185(1):105–157, 2003.

[6] Alexey S. Matveev and Andrey V. Savkin. *Qualitative Theory of Hybrid Dynamical Systems*. Springer, 2001.

[7] Norbert Giambiasi, Bruno Escude, and Sumit Ghosh. GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems. *Transactions of the Society for Computer Simulation International*, 17(3):120–134, 2000.

[8] K. Hopkinson, Xiaoru Wang, R. Giovanini, J. Thorp, K. Birman, and D. Coury. EPOCHS: A Platform for Agent-Based Electric Power and Communication Simulation Built From Commercial Off-The-Shelf Components. *IEEE Transactions on Power Systems*, 21(2):548–558, May 2006.

[9] Anton Cervin and Karl-Erik Årzén. TrueTime: Simulation tool for performance analysis of real-time embedded systems. In Pieter J. Mosterman and Gabriela Nicolescu, editors, *Model-Based Design for Embedded Systems*. CRC Press, 2009.

[10] T. Beltrame. Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems Using DEVS Methodology. Master's thesis, Deparment of Computational Science, ETH Zurich, Zurich, Switzerland, 2006.

[11] D. Frnqvist, K. Strandemar, K.H. Johansson, and J.P. Hespanha. Hybrid Modeling of Communication Networks Using Modelica. In *Proceedings of the 2nd International Modelica Conference*, pages 209–213, Oberpfaffenhofen, Germany, March 2002.

[12] M.A.P. Remelhe. Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment. In *Proceedings of the 2nd International Modelica Conference*, pages 203–207, Oberpfaffenhofen, Germany, March 2002.

[13] P. Mosterman, M. Otter, and H. Elmqvist. Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. In *Proceedings of the 1998 Summer Computer Simulation Conference*, Reno, Nevada, USA, July 1998.

[14] V.S. Prat, A. Uriquia, and S. Dormido. ARENALib: A Modelica Library for Discrete-Event System Simulation. In *Proceedings of the 5th International Modelica Conference*, volume 1, pages 539–548, Vienna, Austria, September 2006.

[15] Hongyan (Bill) Song. Infrastructure for DEVS Modelling and Experiment. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2006.

[16] J.A. Ferreira and J.P. Estima de Oliveira. Modelling Hybrid Systems Using StateCharts and Modelica. In *Proceedings of the 7th IEEE International Conference on Emerging Technologies and Factory Automation*, volume 2, pages 1063–1069, 1999.

[17] M.C. D'Abreu and G.A. Wainer. M/CD++: Modeling Continuous Systems using Modelica and DEVS. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 229–236, September 2005.

[18] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, Sixth Edition.* McGraw-Hill, New York, New York, 2005.

[19] Andrea Balluchi, Luca Benvenuti, Maria Domenica Di Benedetto, Claudio Pinello, and Alberto Luigi Sangiovanni-Vincentelli. Automotive Engine Control and Hybrid Systems: Challenges and Opportunities. *Proceedings of IEEE*, 88(7):888–912, 2000.

[20] Gabrial A. Wainer and Norbert Giambiasi. Cell-DEVS/GDEVS for Complex Continuous Systems. *SIMULATION*, 81(2):137–151, February 2005.

[21] Ernesto Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

[22] J. Frederick Klingener. Combined Discrete-Continuous Simulation Models in ProModel for Windows. In *Proceedings of the 27th Winter Simulation Conference*, pages 445–450, Arlington, VA, USA, December 1995.

[23] J. Frederick Klingener. Programming Combined Discrete-Continuous Simulation Models for Performance. In *Proceedings of the 28th Winter Simulation Conference*, pages 833–839, Coronado, CA, USA, 1996.

[24] Francois E. Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, 2006.

[25] David L. Pepyne and Christos G. Cassandras. Optimal Control of Hybrid Systems in Manufacturing. *Proceedings of IEEE*, 88(7):1108–1123, 2000.

[26] W. David Kelton, Randall P. Sadowski, and Deborah A. Sadowski. *Simulation with Arena.* McGraw Hill, New York, New York, 2002.

[27] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley-Interscience, December 1999.

[28] James Nutaro and Hessam Sarjoughian. Design of Distributed Simulation Environments: A Unified System-Theoretic and Logical Processes Approach. *SIMULATION*, 80(11):577–589, 2004.

[29] H.S. Sarjoughian and B.P. Zeigler. DEVS and HLA: Complimentary Paradigms for M&S? *Transactions of the Society for Computer Simulation International*, 17(4):187–197, 2000.

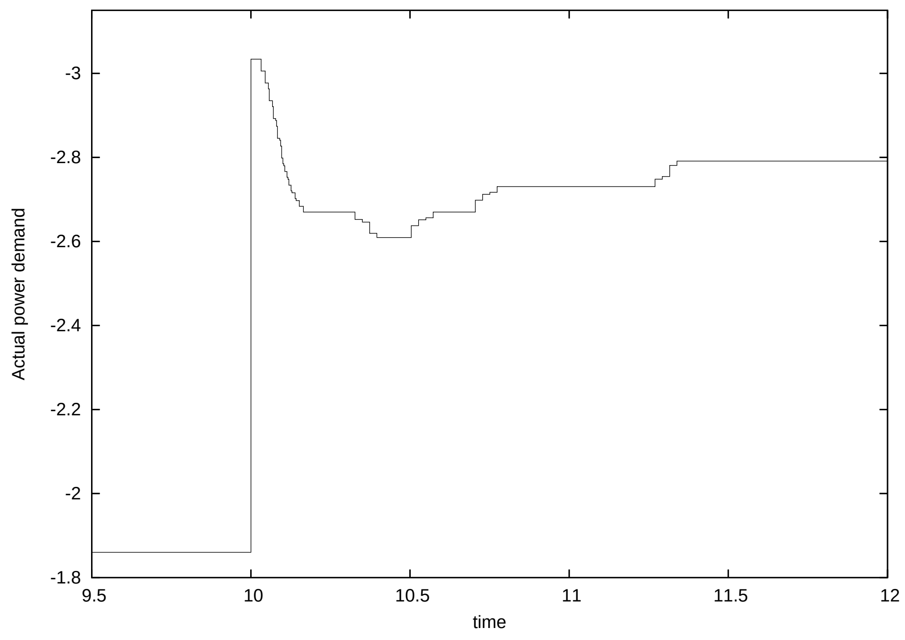
[30] T. Lake, B.P. Zeigler, H.S. Sarjoughian, and J. Nutaro. DEVS Simulation and HLA Lookahead. In *Spring Simulation Interoperability Workshop*, 2000. 00S-SIW-160.

[31] T.E. McDermott, R.C. Dugan, T.L. King, and M.F. McGranaghan. Modelling distribution automation schemes with a control systems overlay. In *IEEE Power & Energy Society General Meeting (PES '09)*, pages 1–3, July 2009.

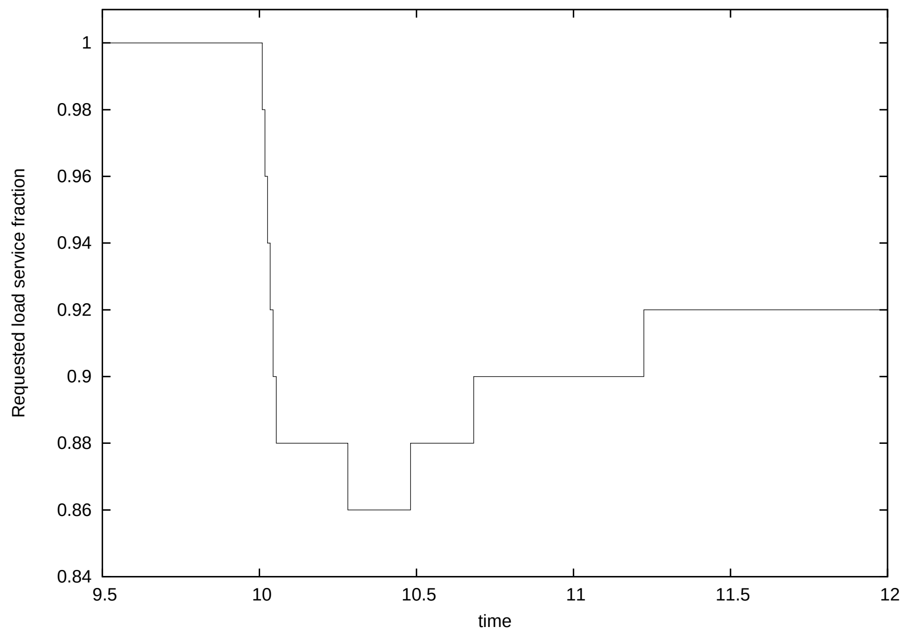
[32] Sara Mullen. Power System Simulator for Smart Grid Development. Master's thesis, University of Minnesota, Minneapolis, MN, May 2006.

[33] Anthony Ralston and Philip Rabinowitz. *A First Course in Numerical Analysis, Second Edition.* Dover Publications, Mineola, New York, 1978.

[34] Andrew S. Tanenbaum. *Computer Networks, Third Edition.* Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.



(a) Actual load



(b) Fraction of load requested

Figure 14: Actual load and load fraction requested during the control action. The jagged load profile is due to delay in the communication network.