Multiscale representation of simulated time

Rhys Goldstein, Azam Khan, Olivier Dalle, Gabriel Wainer

Multiscale Representation of Simulated Time

Rhys Goldstein[1], **Azam Khan**[1], **Olivier Dalle**[2] **and Gabriel Wainer**[3]

Abstract
To better support multiscale modeling and simulation, we present a multiscale time representation consisting of data types, data structures, and algorithms that collectively support the recording of past events and scheduling of future events in a discrete-event simulation. Our approach addresses the drawbacks of conventional time representations: limited range in the case of 32- or 64-bit fixed-point time values; problematic rounding errors in the case of floating-point numbers; and the lack of a universally acceptable precision level in the case of brute force approaches. The proposed representation provides both extensive range and fine resolution in the timing of events, yet it stores and manipulates the majority of event times as standard 64-bit numbers. When adopted for simulation purposes, the representation allows a domain expert to choose a precision level for his/her model. This time precision is honored by the simulator even when the model is integrated with other models of vastly different time scales. Making use of C++11 programming language features and the Discrete Event System Specification (DEVS) formalism, we implemented a simulator to test the time representation and inform a discussion on its implications for collaborative multiscale modeling efforts.

Keywords
Multiscale simulation, time scale, floating-point arithmetic, rounding error, collaborative modeling

1. Introduction

Multiscale modeling and simulation has attracted considerable attention in a number of fields, notably computational biology and materials research where macroscopic transformations routinely emerge from interactions between atoms and molecules. The fact is that most real-world systems have the potential to be studied over a spectrum of scales. Artificial systems, for example, can be viewed as hierarchies of components, and the smallest of these components can be investigated based on their physical and chemical structures. Even if a domain expert's goal is to obtain a simplified single-scale model of the system of interest, multiscale approaches are useful for ensuring the simple model is valid.

A look at past and present multiscale modeling efforts reveals successes and shortcomings, both of which are expressed in a position paper by Hoekstra et al. [1]:

> Multiscale modelling is an actively pursued approach to make sense of wide ranges of phenomena, both natural and anthropogenic. In many different communities, impressive results can be presented. [...] However, in most if not all cases of concern, the research and associated funding to pursue such studies are confined within the boundaries of individual scientific and engineering disciplines. In our view, this renders the field unnecessarily disparate and fragmented. Indeed, it has already led to a slowing down and even stagnation in many relevant topics, to reinventing the wheel, to confusion with respect to terminology and concepts, and to sub-optimal solutions for the implementation of production mode multiscale models running on state-of-the-art computing infrastructures.

According to Hoekstra et al., a single advancement in multiscale modeling is typically confined to a single community working in isolation. The lack of coordination between disciplines leads to the re-invention of similar approaches rather than the enhancement of those approaches. The emergence of inconsistent terminology further complicates the sharing of ideas, making collaboration all the more difficult.

The popularity of multiscale approaches, combined with the fragmented manner in which they are pursued, produces a need for general solutions that can be

[1]Autodesk Research, Canada
[2]University of Nice Sophia Antipolis, France
[3]Carleton University, Canada

Corresponding author:
Rhys Goldstein, Autodesk Research, 210 King Street East, Suite 500 Toronto, Ontario, Canada M5A 1J7
Email: rhys.goldstein@autodesk.com

adopted by researchers across a broad range of disciplines. The idea is to focus on scale-related challenges that re-appear in many fields, and address these challenges in an optimal manner. This paper contributes a general solution to a foundational problem: a flexible computer representation of simulated time. While the representation accommodates nearly all simulations, its distinguishing features specifically target the demands of collaborative, multiscale modeling efforts.

Commonly used computer representations of simulated time, those based on standard 32- or 64-bit fixed-point decimal numbers or binary floating-point numbers, are problematic for multiscale simulation due to the increased potential for numerical errors to alter simulation results. Even with a single scale, the rounding of time values may affect the timing of events to a degree. In some cases, rounding errors may cause events to be reordered[2]. But as we observe in Section 3, the presence of multiple time scales dramatically increases the likelihood and severity of time-related inaccuracies. The underlying issue is that the magnitude of the largest temporal rounding errors is determined by the longest time scales, whereas the tolerance of the simulation to these errors is constrained by the shortest time scales. If the longest and shortest time scales differ greatly, significant distortions in small-scale behavioral patterns are likely to occur.

The multiscale time representation presented in this paper handles rounding errors in a controlled fashion, alleviating many of the unexpected problems that can arise when models featuring different time scales are integrated. The representation takes the form of data types, data structures, and algorithms that collectively support the recording of past events and the scheduling of future events in a discrete-event simulation. The overall approach provides the following benefits:

- SI time units (i.e. seconds, milliseconds, microseconds, etc.) can be represented exactly.
- Time durations, used for modeling, exploit efficient 64-bit operations.
- Time points, used by simulators, provide both extensive range and fine resolution where needed.
- Event handling algorithms introduce no rounding errors despite using mostly 64-bit operations.

When integrating models of vastly different scales, the proposed representation is capable of honoring the specified precision of each model while storing and manipulating the majority of event times as standard 64-bit numbers. To realize these benefits, we embrace the well-established use of *composite models* to define modular or hierarchical structures containing instances of indivisible *atomic models*. We also rely on a core convention of the Discrete Event System Specification (DEVS) formalism[3], the fact that atomic models measure durations relative to the current point in simulated time. Furthermore, each atomic model be sufficiently focused in scale that its time durations can be represented as $m \cdot \delta t$, where m is any integer less than 10^{15} in magnitude and δt is a unit of time precision assigned to the atomic model. The end result is that each atomic model has essentially one time scale, whereas a composite model may span a range of time scales if its component models have different precision levels.

As described in Section 4, our approach exhibits novel features such as the notion of "perceived time", an operation called "multiscale time advancement", and the use of epochs as part of the event scheduling process. Importantly, a typical modeler should be able to enjoy the benefits of the proposed approach without acquiring detailed knowledge of these underlying concepts. Section 5 examines the implications of the multiscale time representation from the modeler's perspective, discusses its implementation in a prototype simulator, and compares experimental results to those obtained with conventional time representations.

One aim of this work is to support modelers who actively pursue multiscale approaches, particularly for applications which involve extremely disparate time scales. Examples of such applications include laser micromachining[4], or the study of protein structural rearrangements[5]. Yet even if domain experts contribute only single-scale models to their respective communities, multiscale issues will emerge from (a) inevitable differences in scale among models and (b) attempts to integrate the models. Efforts to collaborate in the modeling of complex systems will therefore lead communities toward multiscale simulation, creating a demand for general solutions including a common, effective, and complete representation of time.

2. Multiscale modeling and simulation

The 2013 Nobel Prize in Chemistry, awarded for the "*development of multiscale models for complex chemical systems*"[6], draws attention to the importance of accounting for multiple scales. But while publications on multiscale approaches are often specific to one small-scale and one large-scale system within a particular field of study, here multiscale issues are considered from a broader interdisciplinary perspective.

2.1. Scale

Experts often associate a model with an approximate length or time measure referred to as the model's "scale". Some models are considered to have a time scale but no length scale, some have a length scale but no time scale,

some have both length and time scales, and some models are described as having multiple scales in either space or time. Despite the widespread practice of associating models with scales[7], the literature provides little guidance on how a model's scale should be assessed.

We assert that scale is not simply an aggregation of the prominent distances or durations that appear in a model or simulation. Rather, *scale* is an approximation of the degree to which distances or durations must be altered to have an appreciable effect on the implications of a model or the results of a simulation. Suppose that a distance Δx or duration Δt appears somewhere in a modeling project. Perhaps Δx is a model parameter. Perhaps Δt is a duration that tends to re-occur during a simulation. This does not mean Δx or Δt are representative of scale. The question is whether changing distances by roughly Δx, or changing durations by roughly Δt, will effect the outcome of a digital experiment based on the model.

2.1.1. One scale vs. multiple scales

Consider the models illustrated in Figure 1, which involve one or two moving particles of radius r trapped within a compartment of radius R. We assume the particles travel at a constant speed, and that the reflection angle after each inward bounce is random.

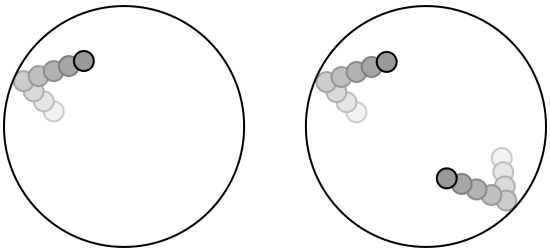


Figure 1. Models featuring one or two small particles of radius r inside a larger compartment of radius R. Both models involve both long and short distance measures (i.e. compartment and particle radii), but not necessarily two distinct scales.

Let us suppose the illustrations in Figure 1 are not to scale, as r is actually a few orders of magnitude shorter than R. Each model therefore features two vastly different distance measurements. Nevertheless, it could be argued that only the model on the right has multiple scales. The reasoning is as follows. The single-particle model on the left of Figure 1 can predict, for example, the average duration between events at which the particle bounces inward off the compartment wall. Although technically this inter-bounce duration depends on both R and r, the particle radius r has a negligible effect on the distance the particle must travel between bounces. (The travel distance will be between R and $2 \cdot R$ for most reflection angles. The particle size reduces this distance by $2 \cdot r$, but we have assumed $r \ll R$.) Were we to increase r by some Δr, then Δr would have to be within roughly an order of magnitude of R to significantly affect the model's predictions. Thus we claim the left model has only a single length scale of approximately the compartment radius. By contrast, the model on the right has an additional ability in that it can predict the frequency of particle-particle collisions. This prediction is affected by changes in r within an order of magnitude of r. Hence the two-particle model has two scales: one associated with the compartment radius and another associated with the particle radii.

The Figure 1 models focus on length scales, but the same line of reasoning applies to time. If a model exhibits a duration Δt, we assess its time scale by asking how much Δt would have to be varied to significantly impact the results. If a model features disparate duration values, it may or may not have multiple time scales.

2.1.2. Time steps vs. time quanta

Time steps and time quanta are of particular importance in the context of simulation models and their scales. These measures relate to the fact that certain instants of simulated time are associated with self-contained computations known as *events*. The fact that certain time points play a prominent role is the basis of a concept called time granularity[8,9,10]. For our purposes, it suffices to focus on two specific concepts related to granularity: time resolution and time precision[11].

Time resolution characterizes the frequency of time points at which similar types of events occur. In discrete-time simulation, time resolution may be expressed using a fixed *time step* that separates consecutive event times. The longer the time step, the coarser the resolution and the larger the discretization errors.

Time precision characterizes the frequency of time points to which event times are rounded. In discrete-event simulation, time precision may be expressed using a fixed *time quantum* that evenly divides all event times. The longer the time quantum, the coarser the precision and the larger the rounding errors.

The scale of a simulation model has important implications pertaining to time resolution and precision. If a model's time resolution is tied to a time step, then lengthening the time step toward the model's time scale will tend to produce noticeable discretization errors that affect the quality of simulation results. Similarly, if a model's time precision is tied to a time quantum, then lengthening the quantum toward the model's time scale will produce significant rounding errors that also impact the quality of the results.

Fortunately, in the case of time precision, it is typically not costly to choose a quantum significantly shorter than the model's time scale. Choosing a fine precision level means allocating more memory to each computed time point, but it should not increase the number of events since event frequency is tied to resolution. Neverthe-

less, modelers may feel compelled to choose excessively coarse precision levels when faced with multiple time scales. It is possible that a long time quantum selected to accommodate a large-scale model could approach the time scale associated with a short-scale model. Such a scenario might lead to rounding errors that adversely affect a modeler's digital experiments.

2.1.3. Scale vs. fidelity

One final observation about our definition of scale is that it excludes the distinct yet important concept of *fidelity*, meaning "level of detail". To illustrate, consider two models of the same electronic circuit. The high-fidelity model includes every resistor, capacitor, transistor, and basic component. The low-fidelity model treats various sections of the circuit as higher-level components, abstracting away the lower-level details. Now assume both models neglect the physical layout of components, so there is no length scale, and both neglect delays in the propagation of voltage changes, so there is no time scale. Combining these models would yield multiple levels of detail, but not multiple scales.

Our interpretation of "scale" and "multiscale" is consistent with the majority of the literature on multiscale approaches, which we review in Section 2.2. It is also a useful interpretation in that discrepancies in continuous quantities of space and time give rise to the issues addressed in this paper. Nevertheless, modeling approaches which address the presence of multiple levels of detail are both relevant and complementary to multiscale modeling. One problem common to both classes of approaches is the introduction of error through the aggregation and disaggregation of data [12]. Whereas this paper focuses strictly on scale, discrepancies in both scale and fidelity are often explored hand-in-hand [13,14].

2.2. Multiscale modeling approaches

Imagine that computational resources were essentially unlimited, and one could therefore perform unfathomably complex molecular dynamics simulations such as one that tracks the location of every atom in a human body. The information provided by such a model might well span close to a dozen orders of magnitude. However, we would not necessarily consider this a "multiscale approach" to modeling, since the entire simulation is based on a single small-scale method.

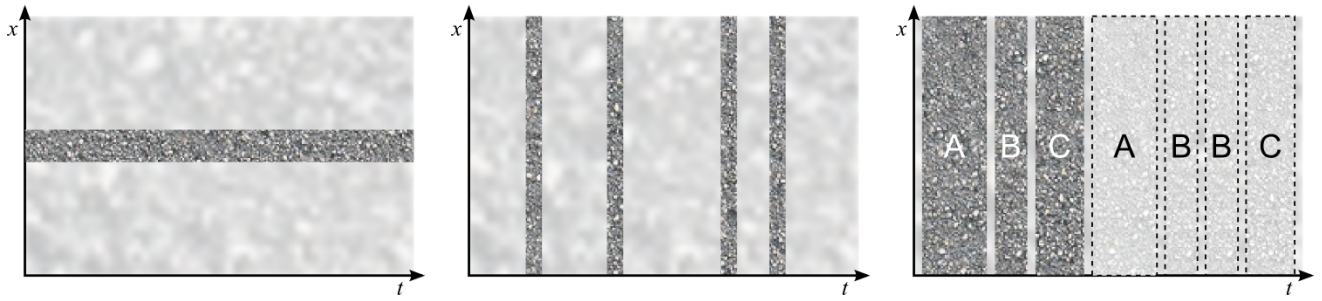
A *multiscale approach* implies that some form of heterogeneity in the representation of a system is based on a discrepancy in scale. More specifically, the model is intentionally designed to exploit some scale-related observation about the represented system, and some benefit is desired as a result. In many cases, the observation involves a small-scale behavior arising only within small regions of space, as in Figure 2a, or during short periods of time, as in Figure 2b; these regions or time periods can be modeled differently from all other regions or time periods. The desired benefits of a multiscale approach may include ease of inputting data or interpreting results. Though in most cases, the primary reason for adopting such an approach is to make more effective use of computational resources.

It is theoretically possible to simulate a wide range of large-scale systems using a brute force application of a small-scale method. The hypothetical application of molecular dynamics to an entire human body is one example. However, many such simulations would remain beyond the capabilities of modern computing technology even if the world's digital infrastructure could be commandeered to execute them. Brute force approaches are limited by two seemingly inescapable trends. First, extending a simulation by a factor of 10 in either time (t) or a single dimension of space (x, y, or z) typically increases the number of operations and possibly also the memory requirements by an order of magnitude or more. Second, one is generally compelled to extend multiple dimensions (e.g. x and y, or x and z and t), as opposed to extending just one dimension by itself (e.g. just x or just t). Thus the computational requirements associated with brute force approaches can be expected to increase by multiple orders of magnitude for every desired 10-fold increase in scale.

Fortunately, complex real-world systems tend to exhibit scale-related characteristics that can be exploited when allocating computational resources. In fact, multiscale approaches are sufficiently popular that three journals are largely dedicated to them: the *Journal of Multiscale Modeling* (World Scientific), the *Journal of Multiscale Modeling and Simulation* (SIAM), and the *Journal of Coupled Systems and Multiscale Dynamics* (American Scientific Publishers). Moreover, any journal related to modeling might feature a few if not many papers describing multiscale modeling efforts. Three specific disciplines feature particularly wide arrays of multiscale approaches: computational biology [15,16], materials science [17], and applied mathematics [18,19]. We list some of the best-known examples from these fields.

2.2.1. Computational biology

Popular among researchers in computational biology, *coarse-graining* methods reduce the complexity of all-atom resolution protein models by treating groups of atoms as elementary particles. The exploited observation about real-world proteins is that certain configurations of nearby atoms undergo comparatively little deformation on the scale of the entire protein molecule. Saunders and Voth [20] emphasize the importance of a formal connection between a coarse-grained (CG) representation and the corresponding all-atom resolution molecular dynamics (MD) model.



(a) Small-scale computations in one region. **(b)** Intermittent small-scale computations. **(c)** Recurring transitions.

Figure 2. Strategies for focusing intensive small-scale computations on subregions (shown as dark, speckled areas) of a large-scale spatiotemporal domain. The domains shown feature one dimension of space on the vertical axis, and time on the horizontal axis.

Although most applications of coarse-graining pertain to molecular dynamics at extremely small scales, the underlying idea of treating several entities as one has been applied to somewhat larger-scale biological processes. Examples include multiscale models of blood clot formation developed using machine learning [21] or graph dynamical systems [22].

2.2.2. Materials science

In materials science, a canonical example of a multiscale approach is the simulation of crack propagation, where the deformation rate is dramatically higher near the crack tip than elsewhere. Abraham et al. [23] model a silicon slab using a coarse finite-element (FE) region, which surrounds a more detailed molecular dynamics (MD) region, which in turn encompasses a growing crack, which in turn features small but highly detailed quantum tight-binding (TB) regions near either end of the crack. Two additional "hand-shaking" regions line the FE-MD and MD-TB interfaces.

Whereas crack propagation simulations are similar to Figure 2a in that they dedicate intensive small-scale computations to particular regions of a spatiotemporal domain, Figure 2c illustrates multiscale approaches based on repetition. Models may have recurring transitions that can be computed once at a small scale, then reapplied as needed in a large-scale simulation. Brereton et al. [24] offer an example. To predict how electrons traverse organic semiconductor materials, matrix calculations produce transition probabilities specific to low energy regions where the electrons become temporarily trapped. These probabilities inform a larger-scale simulation. Note that instead of focusing on temporal transitions, as in Figure 2c, one may exploit repetition over space using representative volumes [25].

2.2.3. Applied mathematics

Multiscale approaches in applied mathematics tend to focus on numerical integration involving vastly disparate rates of change. Equation-free approaches recognize that while the Euler, Runge-Kutta, and other numerical integration methods appear to require a closed-form expression for the derivative of a function, other means of evaluating the derivative may suffice. In particular, the slope of a slowly-varying quantity can be approximated at various points through small-scale simulations, alleviating the need for a differential equation [26]. The Heterogeneous Multiscale Method also uses small-scale computations where needed, but differential equations are closely scrutinized to develop problem-specific integration procedures. Emphasis is placed on the properties of large-scale dynamics that emerge when small-scale fluctuations are averaged out [27].

When classifying an approach as multiscale, an important caveat is that the strategy for exploiting a scale-related observation does not entail the elimination of all but one scale from a modeling effort. For example, a purely coarse-grained protein simulation does not constitute a multiscale approach if the atom scale is neglected; rather the CG simulation must be informed in some way by all-atom models. In light of this caveat, multiscale approaches generally retain each relevant scale in one form or another. An exception to this rule is the seamless multiscale method, where for favorable differential equations the smaller of two time scales is effectively replaced with an intermediate scale [28]. In this case the two original scales are not both retained, but the model resulting from the transformation still features two distinct scales. The seamless method has been generalized to address three or more relevant scales [29].

2.3. Collaborative multiscale modeling

As mentioned, a multiscale approach to modeling implies that heterogeneity in the representation of a system is based on a discrepancy in scale. Generalizing the concept, we suggest that *systems science* arises from the introduction of any kind of heterogeneity into the representation of any system. The presence of multiple scales

is one aspect of a system that may serve as a fundamental basis for heterogeneity [1], though there are complementary aspects which may deserve equal or greater attention depending on the challenge at hand. In the case of *multi-domain modeling* [30], heterogeneity in representation is based on the relevance of multiple domains, as in multiple types of systems. Similarly, *multi-paradigm modeling* [31] implies heterogeneity based on the utility of multiple paradigms, as in multiple sets of techniques and conventions for defining models.

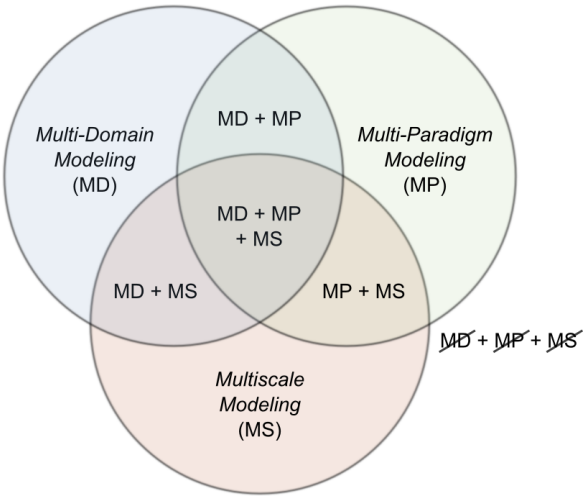


Figure 3. Intersecting of multi-domain, multi-paradigm, and multiscale modeling.

The fact that a diverse set of modeling efforts may incorporate multiple domains, multiple paradigms, and/or multiple scales leads to the classification scheme illustrated in Figure 3. The idea is that any such effort can be categorized into one of the eight sections formed by the overlapping circles. There are separate sections for purely multi-domain, purely multi-paradigm, and purely multiscale efforts, and one outer section for efforts involving a single domain, paradigm, and scale.

Multi-domain, multi-paradigm, and multiscale modeling often coexist. In fact, a single form of heterogeneity in representation can be associated with more than one of these aspects. The intersection of two or three of these aspects is emphasized by the inner regions of Figure 3 where the circles overlap. If two domains are relevant, for example (a) the behavior of humans and (b) their impact on food and water resources, they may be modeled using distinct paradigms such as (a) agent-based simulation and (b) system dynamics; such scenarios fall into the MD + MP category. Groen et al. [32] highlight the MD + MS category by reviewing multiscale modeling in conjunction with multiphysics applications, a subcategory of multi-domain modeling. A focus on MP + MS can be found in Vangheluwe et al. [31]. They list levels of abstraction, which overlaps with multiscale modeling, alongside multi-formalism modeling and meta-modeling as key research directions in multi-paradigm modeling. The close relationships among the three aspects of modeling are evident in titles such as "*Multi-scale and multi-domain computational astrophysics*" [33], "*Multi-paradigm multi-scale simulations for fuel cell catalysts and membranes*" [34], and "*Multiscale coupling and multiphysics approaches in earth sciences*" [35].

Collaborative modeling can be promoted by supporting the integration of models that differ in domain, paradigm, and/or scale. Such efforts naturally lead to multi-domain, multi-paradigm, and/or multiscale models, creating a need for general modeling solutions. For multiple domains, one general solution is provided by equation-based modeling, exemplified by the Modelica language [30]. For multiple paradigms, a core technique is the transformation of various formalisms onto a single formalism such as DEVS [31]. For multiple scales, progress has been made in the form of a multiscale modeling language and framework by Chopard et al. [36], which emphasizes scale-related strategies comparable to those shown in Figure 2. There are also formalism extensions such as Multi-Level DEVS [37] and downward/upward atomic models [38], which facilitate the transformation of data and time granularity among levels in a model hierarchy. Yet little attention is given to multiscale computer representations of space and time. Whereas the three dimensions of space give rise to a multitude of possible representations, the single dimension of time presents its own challenges and opportunities. As mentioned in Section 2.1, a modeler may choose an excessively coarse precision level to accommodate the longest of several time scales. Temporal rounding errors may then approach the shortest time scale, affecting simulation results. We address this challenge, aiming to remove a barrier to collaboration on multiscale modeling efforts.

3. Conventional time representations

Collaborative model development currently entails the following dilemma: the larger the community of modelers, the greater the need for all modelers to adhere to the same modeling conventions, but the harder the task of choosing conventions that meet all modelers' needs. Here we explain why the most commonly used representations of simulated time may fail to meet the needs of large communities of collaborating modelers, particularly if their models differ significantly in scale.

3.1. Fixed-point time representation

A *fixed-point time representation* implies that some time quantum δt is common to an entire computational process such as a simulation run, and any finite time value in the process can be expressed as $m \cdot \delta t$ for some integer m. This is an extremely common representation of

time in computer applications, particularly if δt is 1 s, 1 ms, 1 μs, or 1 ns. Discrete-time simulations often adopt this approach with δt being the time step and m being the number of processed iterations. When discrete-event simulations use fixed-point time values, the time quantum serves as a common factor of all durations between events. A suitable common factor can often be derived from a model specification [11], though in practice it is usually chosen arbitrarily. Let us limit our discussion to fixed-point decimal representations, such that δt is 10^d seconds for some potentially negative integer d. This measure is also known as time precision; a larger δt is associated with a coarser level of precision. We assume m is represented as a 32-bit or 64-bit two's complement number, which means that its range is either $-2^{31} \leq m < 2^{31}$ or $-2^{63} \leq m < 2^{63}$.

The first problem with 32- or 64-bit fixed-point decimal representations of time is the limited range they provide. This is not typically a problem for discrete-time simulation, as even the 32-bit option accommodates over 2 billion iterations. For discrete-event simulation, however, convention suggests that many of these bits would be dedicated to achieving a precision level much finer than the time scale of the model. Part of the motivation behind discrete-event simulation is to treat time as a continuous quantity. Rounding all time values to a coarse precision level would undermine this benefit.

To illustrate range limitations, consider the simulation of a laser micromachining process. As indicated by Gattass and Mazur [4], the duration of each laser pulse can be as short as 10 fs whereas the resolidification period that follows can persist beyond a microsecond. If time is represented using 32-bit fixed-point time values and a 1 fs time quantum, simulations are limited to just over 2^{31} fs, or 2 μs. This is barely sufficient for a single pulse/resolidification cycle. Furthermore, one must verify that temporal rounding errors on the order of a femtosecond will not invalidate the simulation of small scale effects during the pulse. However 64-bit time values allow one to use a 1 ys time quantum, improving precision by a factor of 1,000,000 and permitting 10 ms simulation runs that accommodate many pulses. It is clear that only the most ambitious multiscale models would challenge the range of a 64-bit integer. Unfortunately, limited range is not the only problem with fixed-point time values.

A fixed-point representation of time raises the issue of how a common precision level is selected, and by whom. The simplest and most computationally efficient option is to hardwire the precision, ensuring consistency. One of many examples of this approach can be found in the `time` package of the Go programming language. The `Duration` type is a 64-bit fixed-point time representation with a 1-nanosecond time quantum, which limits the maximum duration to roughly 290 years [39]. One expects this data type to be reasonable for most human-scale simulations, but inappropriate for small-scale molecular dynamics simulations or large-scale computational astrophysics simulations. Another option is to allow users to select the precision level prior to a simulation run. The OMNeT++ simulation framework provides a global configuration variable that specifies 1 s, 1 ms, 1 μs, 1 ns, 1 ps, 1 fs, or 1 as precision [40]. The ns-3 simulator uses a similar approach, adding 1 min, 1 hr, 1 day, and 1 yr to the list of allowable precision levels [41].

Even if the common time precision is customizable, as it is in OMNeT++ and ns-3, a problem remains in that a model's behavior may unexpectedly change when a different precision level is chosen for the encompassing simulation. For example, consider a model featuring a (1/3) s delay. Suppose that temporal errors of around a microsecond are tolerable, so the model requires a time quantum of 1 μs or shorter. If one simulates this model with a 1 μs time quantum, the delay is 333,333 μs. But if the time quantum is later changed to 1 ns, the delay becomes 333,333,333 ns leading to slightly different results, which may be unexpected.

In summary, the limited range of fixed-point time representations can make it difficult to choose an appropriately fine precision level while allowing for sufficiently long simulation runs. A 64-bit multiplier offers a fair degree of flexibility in this regard, but challenges arise in selecting a time quantum common to a set of integrated models. A final drawback with the fixed-point option is the lack of a convenient way to represent positive and negative infinity, though this is addressed by an implementation strategy discussed in Section 3.4.

3.2. Floating-point time representation

A *floating-point time representation* implies that the duration between a representable time point t and the next largest representable time point t' is not a fixed time quantum but rather scales with t. Thus while the relative error may be bounded, the absolute error from operating on a time value will tend to be proportional to the value itself. Conceptually, floating-point numbers consist of a common positive integer base β, a value-specific integer coefficient c, and a value-specific integer exponent γ such that the represented quantity is the real number $c \cdot \beta^\gamma$. Smaller bases are associated with smaller rounding errors [42], so binary floating-point numbers with $\beta = 2$ prevail. Unless otherwise stated, we associate "floating point" with the IEEE 754 double precision standard, a 64-bit binary floating-point representation that allocates 11 bits to the exponent and 53 bits to the coefficient including its sign. IEEE 754 dedicates certain 64-bit sequences to special values such as infinity and NaN (Not a Number), and incorporates these values into its rules for mathematical operations.

The obvious advantage of floating-point time values is the extraordinary range they provide. To fully appre-

ciate this range, let us consider the most extreme time scales under scientific investigation.

At the smaller end of the spectrum, molecular dynamics is among the more popular classes of simulation techniques. MD models generally rely on time steps on the order of 10 femtoseconds[43]. For example, 2.5 fs time steps were used for a number of state-of-the-art all-atom MD simulations performed on the special-purpose Anton 2 supercomputer[44]. In the lesser known field of quantum chromodynamics, time is often measured in fm/c, or roughly $3.34 \cdot 10^{-24}$ s, the time required for light to travel one femtometer[45]. One of the shortest durations in the literature is known as the Planck time. A theory has been proposed that physical time does not advance continuously but rather "ticks" forward by this quantum-like duration[46]. The Planck time is just under 10^{-43} seconds, which does not even approach the lower limit of a floating-point number. Double precision values can be smaller than 10^{-307}, or less than 10^{-323} if one considers subnormal values that become available when the 11-bit exponent reaches its most negative value.

At longer time scales, the Illustris Project is a noteworthy effort in which a 13-billion-year simulation tracks the cosmic evolution of the Universe from shortly after the Big Bang until roughly the present day[47]. If we turn our attention from the origins of the universe to astrophysicists' predictions of its fate, even longer time scales are discussed. After 10^{38} years, the only remaining "stellar-like objects" will be black holes, and around 10^{100} years all protons in the universe will have decayed and all black holes will have "evaporated"[48]. To simulate the demise of all black holes in the universe, one would require a time representation with an upper limit of around 10^{108} seconds or higher. Double precision time values meet this requirement with ease, as they can reach just over 10^{308} s.

It is difficult to imagine a useful simulation that would challenge a 64-bit floating-point time value's range. Precision, however, is a different matter. The 53-bit coefficient offers 53 bits of precision, after one first subtracts the sign bit and then adds the hidden bit that is assumed to be 1 for normalized numbers. These 53 bits translate to about 16 decimal digits, which may seem decent. But the practical consequence is that models should only be coupled, arguably, if they differ by no more than 6 orders of magnitude in time scale. For example, a nanosecond-scale model can be integrated with a millisecond-scale model, but not a second-scale model. The reason why we start with 16 orders of magnitude, but end up with 6, is that 6 orders are lost if one wishes to keep rounding errors below one millionth of the shorter time scale, and another 4 are lost if the simulation is to progress 10,000 times the longer time scale. A desire for longer simulation runs would further detract from the allowable difference between the two scales.

A lack of precision is cited as one of the reasons that floating-point time values were factored out of OMNeT++. The following is from Varga[40]:

> Why did OMNeT++ switch to `int64`-based simulation time? `double`'s mantissa [the coefficient, excluding the sign and hidden bits] is only 52 bits long, and this caused problems in long simulations that relied on fine-grained timing, for example MAC [media access control] protocols. Other problems were the accumulation of rounding errors, and non-associativity (often $(x + y) + z \neq x + (y + z)$, see Goldberg[42]) which meant that two `double` [64-bit floating-point number] simulation times could not be reliably compared for equality.

The latter half of Varga's quote highlights another significant problem with floating-point time values: the fact that rounding error is introduced as a result of addition and subtraction operations. Both fixed- and floating-point representations incur rounding errors for other operations such as multiplication and division. Yet addition and subtraction errors are particularly problematic in the case of simulated time, for several reasons. First, the addition and subtraction of time values are extremely common operations in model code, so these types of rounding errors will accumulate quickly. Second, simulation development frameworks such as OMNeT++ separate user-defined models from a common model-independent simulator; a typical simulator adds and/or subtracts time values, and will therefore impose unavoidable errors on the user if these operations are not exact. Note that general-purpose simulators rarely multiply or divide time values as part of the simulation process. Third, a modeler with a basic understanding of digital technology should expect rounding errors in multiplication and division, but he/she might expect addition and subtraction to yield exact results.

There is yet another unfortunate side effect of floating-point time representations, one particularly relevant to discrete-event simulation. Simulations typically feature a current time variable t that repeatedly advances, mimicking the progression of physical time. Strangely, if t is represented as floating-point number, then rounding errors will tend to increase as the simulation progresses. To understand why this happens, consider the common operation in (1), where Δt is any positive duration added to the current time t to yield a future time t'.

$$t' = t + \Delta t \tag{1}$$

The Δt values produced by a model often vary over simulated time, but rarely trend upward or downward. Hence they become progressively shorter relative to the advancing t, and therefore temporal rounding errors resulting from $t+\Delta t$ operations tend to worsen as a simulation progresses. Particularly troublesome are situations

in which t and Δt become so disparate that t' is rounded down to t, and events that should have been scheduled for different time points instead occur at a common instant. Effectively, the duration Δt is rounded to zero.

Multiscale approaches dramatically increase the risk of positive durations being rounded to zero. Imagine a case where two separate teams develop two distinct models, both relying on floating-point time values. The first team produces a large-scale model, which performs accurately because Δt is never short. The second team works on a small-scale model, which also proves accurate because t never grows long. The models are integrated and tested for a short duration of simulated time. Now t progresses quickly due to the large-scale model, causing some rounding error to emerge in the small-scale model's durations. But the errors are small and go unnoticed. The multiscale model, considered valid, is then deployed for scientific investigation. Longer simulations are conducted in which t becomes very large relative to the small-scale model's Δt values. At some point these durations round to zero and severely undermine the quality of the results.

It is worth acknowledging that there is a well-known technique for coping with floating-point rounding errors. Instead of comparing two values x and y for equality (i.e. $x = y$), one tests whether their difference lies within some small, arbitrary epsilon value ϵ. If $|x - y| < \epsilon$, the assumption is made that rounding error alone is responsible for the difference between x and y, and hence the values should be treated as equal. In the context of simulated time, the interpretation is that two or more events with slightly different time points should be treated as occurring at the same instant. Unfortunately, it is difficult to ensure that every simulation-based experiment uses an appropriate ϵ, particularly if multiple scales are involved. Furthermore, the effect of any ϵ will change over the course of a simulation run as t increases relative to the Δt values. Finally, a choice of ϵ will typically be based on assumptions, and it is difficult to ensure every assumption remains valid when a model is modified or reused by other developers.

A number of parameters similar to epsilon values have been proposed in a simulation context for specific purposes. Wieland [49]'s *threshold of simultaneity* δ can be viewed as an epsilon value that provides an upper limit on randomly generated event time offsets. The purpose of these offsets is to avoid any arbitrary ordering of simultaneous events. Zeigler et al. [50] propose a *time granule* d that improves the performance of synchronous parallel simulation algorithms by treating nearly simultaneous events as simultaneous. Although Wieland's δ and Zeigler's d offer statistical and performance-related benefits for certain applications, they do not provide a general solution to the drawbacks of floating-point time representations. Even with δ or d, one may still encounter the above-stated problems associated with epsilon values.

To summarize, the excellent range of a floating-point time value is counterbalanced by its limited precision, while serious problems result from rounding errors introduced by addition and subtraction. The fact that rounding errors increase as a simulation progresses means that harmful effects may not surface until a model has been tested and put to use. Another disadvantage of binary floating-point time values is that fractional SI time units, such as milliseconds or microseconds, cannot be exactly represented. These units are important due to their popularity as a means of specifying duration parameters. Most of these drawbacks have previously been identified [2]. The implications we most wish to emphasize are those pertaining to multiple time scales, which dramatically increase the chance that floating-point rounding errors will approach the smallest time scale in magnitude. The potential impact of these scale-related effects is revealed in the following section.

3.3. Experimentally observed impact of fixed- and floating-point time representations

To investigate time representations, we introduce a multiscale model inspired by earthquake warning systems used in Japan, Mexico, and California [51]. By detecting P waves, such systems predict the arrival of the subsequent S waves which cause most of the devastation. Although the P waves may precede S waves by only tens of seconds, this advance notice can be used to decelerate trains in anticipation of a possible derailment, or to encourage individuals to seek protection from potential falling objects. One plausible reason to simulate earthquake warning systems is to evaluate their performance and choose the best possible design. However, our interest lies not in the application but rather in the discrepancy among its relevant time scales: the multi-year periods between earthquakes, the tens of seconds between P and S waves, and the much shorter time durations associated with seismic noise.

We note that our model is only loosely inspired by earthquake warning systems. The signal processing involved in the detection system is greatly simplified. We also generalize the model to dissociate it from any particular time scale. In place of earthquakes, our model involves a more general type of undesirable event called an *incident*. Each incident is preceded not by a P-wave, but by a general type of informative event called an *occurrence*. We model a prediction system that detects an occurrence and predicts whether the subsequent incident is life-threatening or benign. The performance of the system is intended to improve over time, since it learns from past occurrences and their associated incidents. The key elements of the prediction system are illustrated at multiple scales in Figure 4.

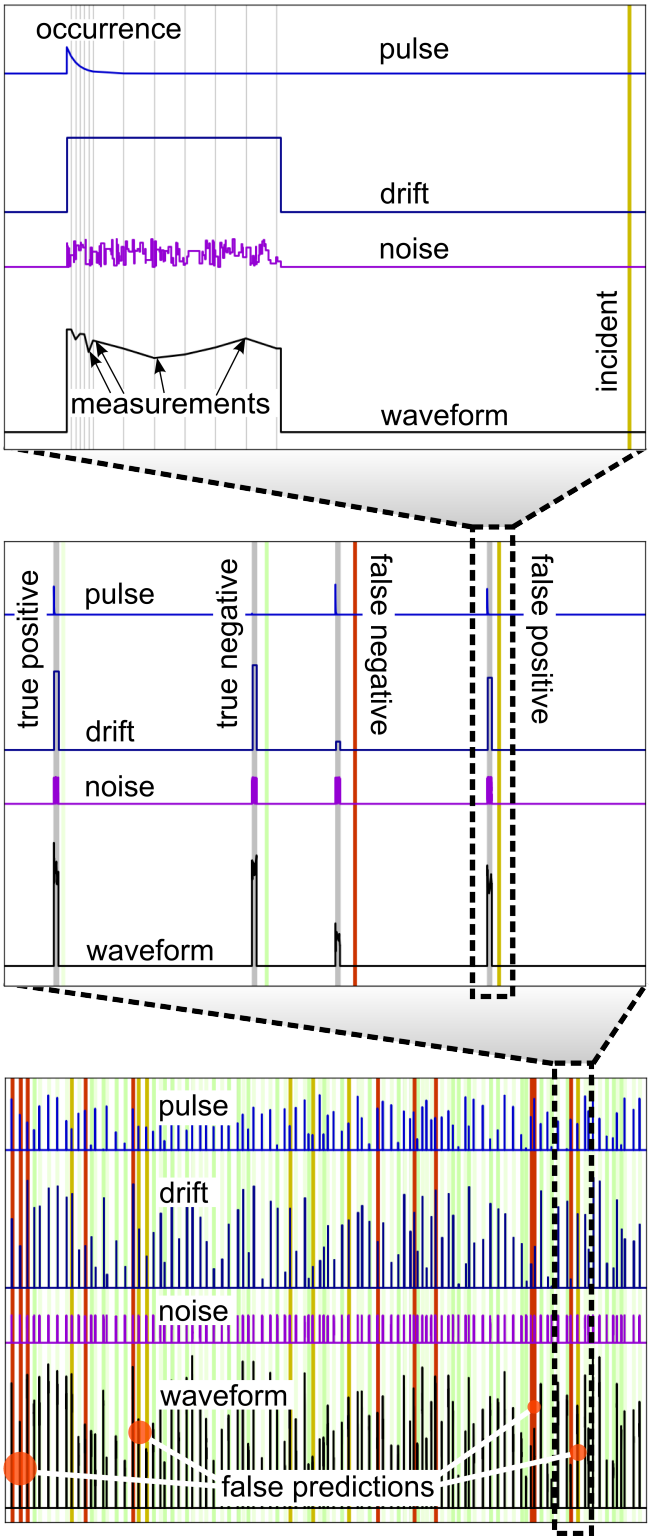


Figure 4. A simulation of a prediction system that learns over time, plotted at different time scales. The upper plot shows one prediction, revealing measurements of an informative pulse obscured by a drift and noise. The middle plot shows four of these predictions. The lower plot covers the full simulation period. Dark vertical lines (red and gold) indicate false predictions, several of which are emphasized at the bottom.

The upper plot in Figure 4 focuses on a single occurrence and its subsequent incident. As shown at the top of this plot, an occurrence is associated with an instantly appearing and rapidly decaying pulse. Larger pulses suggest life-threatening incidents. The magnitude of each pulse is obscured by a low-frequency drift represented as a constant offset, as well as high-frequency noise. The waveform at the bottom is the superposition of the pulse, drift, and noise, sampled at 12 measurement points. Note that the drift and noise are assumed to be present at all times, but are only simulated over periods encompassing these measurements.

Taking a closer look at the top of Figure 4, the 12 measurements include 6 at a high sampling rate followed by another 6 at a low rate. The first 6 are averaged to filter out noise and aggregate the pulse in combination with the drift. The next 6, which are also averaged to reduce noise, aggregate the drift alone. By subtracting the average of the latter 6 measurements from the average of the first 6, one obtains a feature value that can be used to predict whether the impending incident is life-threatening or benign. The feature is compared with a threshold, which is always the midpoint of two averages: the average feature values of past life-threatening and benign incidents. The system learns by simply recalculating this threshold after every incident.

To evaluate the accuracy of the prediction system, we categorize each prediction as follows.

- **True positive**: the incident is predicted to be life-threatening, which turned out to be correct
- **True negative**: the incident is predicted to be benign, which turned out to be correct
- **False positive**: the incident is predicted to be life-threatening, which turned out to be wrong
- **False negative**: the incident is predicted to be benign, which turned out to be wrong

The middle plot in Figure 4 expands the view to encompass four predictions, one of each type. A true positive, such as the first prediction, is shown using a faint green line to mark the incident. A true negative, such as the second prediction, is indicated by a light green color. The third prediction is a false negative, shown using dark red. The last of the four predictions is a false positive, indicated by a gold-colored incident.

The lower plot in Figure 4 covers the full time period of the simulation run, which encompasses exactly 100 predictions. Because the prediction system learns over time, the likelihood of a true prediction should gradually increase. This trend of increasing accuracy can be seen in the plot, notwithstanding a couple prominent clusters of false predictions toward the end of the simulation.

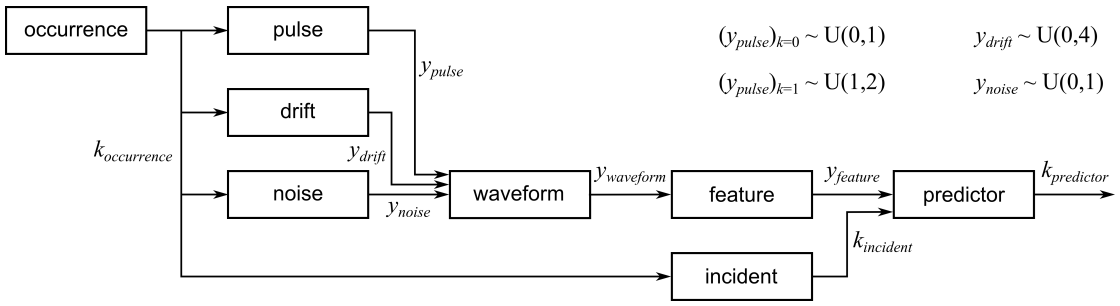


Figure 5. Composition of the prediction system model. The pulse height for benign ($\langle y_{pulse} \rangle_{k=0}$) and life-threatening ($\langle y_{pulse} \rangle_{k=1}$) incidents, the drift height (y_{drift}), and the noise height (y_{noise}) are sampled from the uniform distributions at the top right.

Instead of using time scales associated with earthquakes, we test a variety of relatively long *average occurrence durations* ($\Delta\bar{t}_{occurrence}$) between occurrences and relatively short *average incident durations* ($\Delta\bar{t}_{incident}$) between occurrences and their subsequent incidents. These two duration parameters are each varied by several orders of magnitude. For every combination of $\Delta\bar{t}_{occurrence}$ and $\Delta\bar{t}_{incident}$, the simulation is repeated 1000 times and the average accuracy of the system is reported as the ratio of true predictions (true positives plus true negatives) to total predictions. It should be noted that for this experiment, the purpose of the model is not to maximize the reported accuracy. Rather, the goal is to simply report accuracy values that reliably capture the performance of the system being modeled.

The prediction system is modeled as a composition of eight submodels, connected as shown in Figure 5. At intervals averaging $\Delta\bar{t}_{occurrence}$ in duration, the occurrence instance randomly generates positive ($k = 1$) or negative ($k = 0$) messages with equal probability. These messages trigger a succession of events in the downstream components. The pulse, drift, and noise instances each randomly generate y-values, which are superimposed in waveform to produce the 12 measurements. Each measurement is communicated as a separate $y_{waveform}$ message. All 12 such messages are processed by feature to produce $y_{feature}$, the average of the first 6 measurements minus the average of the latter 6. The predictor instance receives this feature value and immediately predicts the class k. The prediction is either confirmed or contradicted some time later, when incident finally outputs the same message it received earlier from occurrence.

The model is simulated for 100 combinations of two duration parameters: 10 values for each parameter, with a 10-fold gap between successive values. The large-scale average duration between occurrences is varied from 10 seconds to 10^{10} seconds (≈ 317 years). The small-scale average duration between an occurrence and its subsequent incident is varied from 1 second to 1 nanosecond.

$$\Delta\bar{t}_{occurrence} \in \{10^1\,\mathrm{s}, 10^2\,\mathrm{s}, \ldots, 10^{10}\,\mathrm{s}\}$$

$$\Delta\bar{t}_{incident} \in \{10^0\,\mathrm{s}, 10^{-1}\,\mathrm{s}, \ldots, 10^{-9}\,\mathrm{s}\}$$

The actual occurrence and incident durations are randomly generated from uniform probability distributions according to the formulas below.

$$\Delta t_{occurrence} \sim \mathrm{U}(\tfrac{1}{2}, \tfrac{3}{2}) \cdot \Delta\bar{t}_{occurrence}$$

$$\Delta t_{incident} \sim \mathrm{U}(\tfrac{1}{2}, \tfrac{3}{2}) \cdot \Delta\bar{t}_{incident}$$

A number of shorter durations are defined below. All are constants, except for the actual noise segment duration which is sampled from an exponential distribution. The use of fractions $\frac{1}{3}$ and $\frac{1}{7}$ is based on a concern that factors of $\frac{1}{2}$ or $\frac{1}{10}$ might be seen as favoring, respectively, binary floating-point time values or fixed-point decimal time values.

$$\left.\begin{aligned} \Delta t_{predictor} &= \tfrac{1}{3} \cdot \Delta\bar{t}_{incident} \\ \Delta t_{sparse} &= \tfrac{1}{7} \cdot \Delta t_{predictor} \\ \Delta t_{decay} &= \tfrac{1}{3} \cdot \Delta t_{sparse} \\ \Delta t_{dense} &= \tfrac{1}{7} \cdot \Delta t_{sparse} \\ \Delta\bar{t}_{noise} &= \tfrac{1}{3} \cdot \Delta t_{dense} \\ \Delta t_{noise} &\sim \mathrm{Exp}(\Delta\bar{t}_{noise}) \end{aligned}\right\} \begin{array}{l} \text{encompasses all} \\ \text{measurements} \\ \text{separates sparse} \\ \text{measurements} \\ \text{pulse decay} \\ \text{time constant} \\ \text{separates dense} \\ \text{measurements} \\ \text{average noise} \\ \text{segment duration} \\ \text{actual noise} \\ \text{segment duration} \end{array}$$

Observe that $\Delta t_{predictor}$ deliberately encompasses all 12 measurements with time to spare. The 6 high- and 6 low-frequency measurements require a total duration of $6 \cdot \Delta t_{dense} + 6 \cdot \Delta t_{sparse}$, which is equal to $7 \cdot \Delta t_{sparse} - \Delta t_{dense}$, which equals $\Delta t_{predictor} - \Delta t_{dense}$. Thus the multiscale modeling technique of simulating the drift and noise for durations of only $\Delta t_{predictor}$ should have no effect on the results of interest.

The experiment is conducted using a general-purpose discrete-event simulation library modified to support both fixed- and floating-point time representations. For each representation, 1000 repetitions are performed for all 100 combinations of $\Delta\bar{t}_{occurrence}$ and $\Delta\bar{t}_{incident}$.

For 32- and 64-bit fixed-point time representations, the simulations use the shortest base-1000 SI time unit

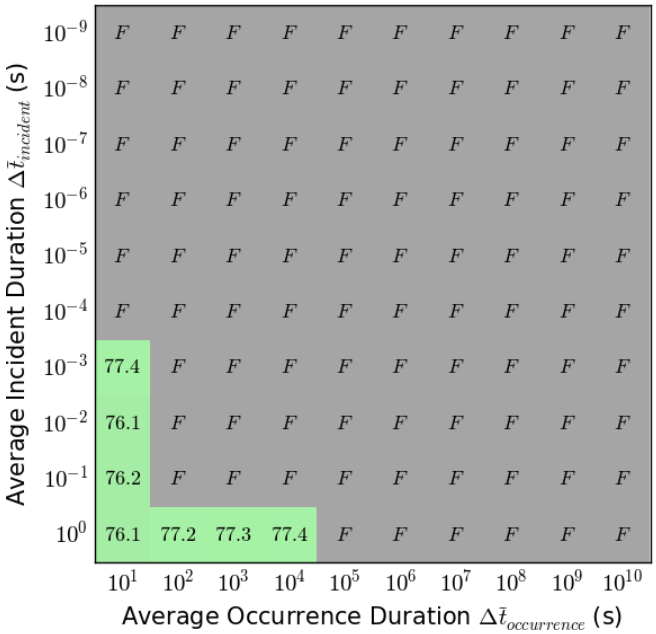


Figure 6. Mean accuracy results using 32-bit fixed-point time values. A value near 76 is considered an acceptable result. An *F* indicates a failed simulation.

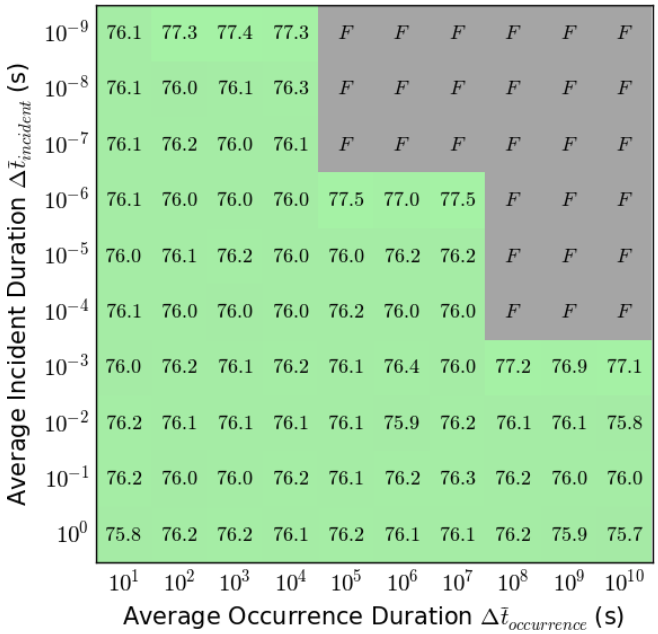


Figure 7. Mean accuracy results using 64-bit fixed-point time values. With 64 bits instead of 32, a far greater separation between scales is accommodated. Yet extremely diverse scales still lead to failure.

δt that can accommodate the maximum duration of each simulation run, roughly $101 \cdot \Delta\bar{t}_{occurrence}$. The formula for δt is given in (2), where *nbits* is either 32 or 64.

$$\delta t = 1000^{\lceil log_{1000}(101 \cdot \Delta\bar{t}_{occurrence}/2^{nbits-1})\rceil} \quad (2)$$

The use of base-1000 SI time units is intuitive to modelers and consistent with OMNeT++ and ns-3. A simulation run is considered to have failed if its shortest duration constant, the average width of a noise segment $\Delta\bar{t}_{noise}$, rounds to zero.

Figure 6 shows the results of the 10 by 10 configurations simulated using 32-bit fixed-point time values. The accuracy values of roughly 76 or 77 are the averages over 1000 repetitions of the number of true predictions out of 100 in each repetition. Based on results obtained from two different simulators and four different time representations, we consider the prediction system's inherent accuracy to be slightly over 76%. Our model is designed such that the system's accuracy does not depend on the times scales used. This characteristic allows us to attribute any variability in the results to errors caused by the time representation. In the case of a 32-bit fixed-point representation, only 3 out of 100 configurations yield the desired value, and only 7 of 100 permit the simulation to complete. In the modern era of 64-bit computing, one has little need to restrict a time value to 32 bits. Yet we include these results to highlight the severely limited range of this representation.

Figure 7 provides the accuracy results for 64-bit fixed-point time values. Doubling the number of bits, a considerably greater fraction of the test cases complete. The mean accuracy is roughly 76%. The variability is consistent with the standard error values we obtained, which were slightly under 0.14 for all 73 successfully simulated configurations. However, 9 particular configurations result in mean accuracies between 76.9% and 77.5%. These outliers are conspicuously situated adjacent to failed configurations, suggesting that computational problems emerge when a simulation approaches the upper limit of a fixed-point time representation.

As mentioned in Section 3.2, a floating-point time representation alleviates the need for a time quantum and practically eliminates restrictions on range. But as evident from the accuracy values in Figure 8, floating-point time values can have a severe and seemingly erratic effect on simulation results. The general trend in Figure 8 can be summarized as follows. The lower left majority of the configurations feature a relatively small range of time scales, and their reported accuracies of around 76% are consistent with one another and with the fixed-point results. Approaching the top right corner of the plot, time scales grow further apart and accuracy values begin to decrease. The accuracies dip below 60% in some cases before dramatically increasing toward a plateau of about 96.5%. Aside from this trend of decline and increase toward the top right corner, no definitive patterns can be found. This suggests that many factors contribute to the overall impact of floating-point time computations.

As one scans Figure 8 from bottom left to top right, the first statistically significant sign of error is the ac-

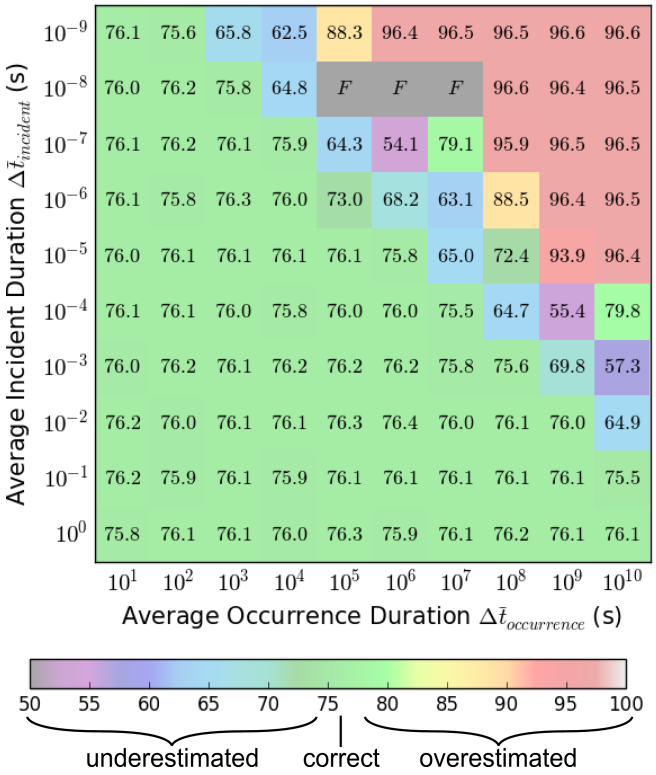


Figure 8. Mean accuracy results using 64-bit floating-point time values. The results become erratic as the time scales diverge.

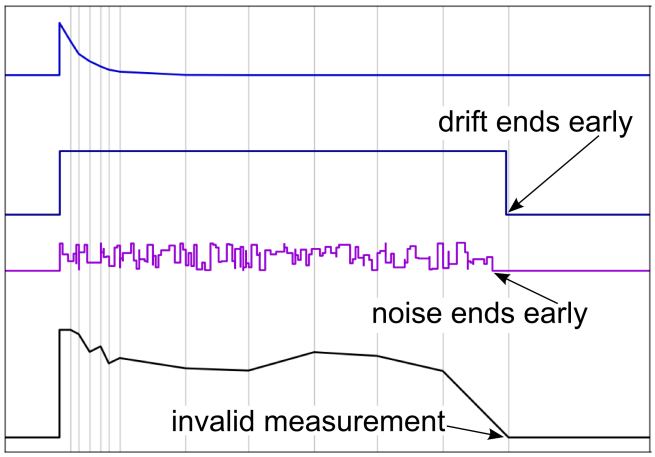


Figure 9. Impact of floating-point time values at $\Delta\bar{t}_{occurrence} = 10^{5}$ s and $\Delta\bar{t}_{incident} = 10^{-6}$ s. Both drift and noise signals end before the last measurement, rendering it invalid.

curacy value of 73.0% near the center of the plot. To understand why the accuracy is underestimated here, we examine a single simulation run. Figure 9 reveals two issues in the last of the run's 100 predictions. Recall that noise and drift are simulated over a duration of $\Delta t_{predictor}$, which theoretically encompasses all measurements with time to spare. Yet here the noise signal ends noticeably early, and on close examination the drift signal ends just slightly before the last measurement. The reason why these signals end prematurely relates to the discrepancy between the increasing current time t and the comparatively stable durations Δt added to it. Rounding errors in $t + \Delta t$ expressions grow with t, and eventually become large relative to Δt. As explained in Section 3.2, the problem worsens as disparate time scales are incorporated into a model.

For the noise signal, individual noise segments are generated until their combined duration reaches $\Delta t_{predictor}$. However, when the individual segment durations are added to the current time t in the simulator, t advances by less than $\Delta t_{predictor}$. The fact that the noise signal is shortened, rather than lengthened, is likely due to short noise segments being rounded to zero.

In the case of the drift, a close investigation reveals that the signal is actually lengthened beyond $\Delta t_{predictor}$ due to rounding error. Unfortunately, the measurement durations are also rounded upward, and to such an extent that they surpass the end of the drift signal. This causes the drift to be excluded from the final measurement. This unexpected behavior occurs toward the end of a simulation run, after the prediction system is mostly trained. The result is an increase in false predictions, and an underestimated accuracy of 73.0%.

Could the problems seen in Figure 9 be fixed with a more robust implementation of the model? The answer is yes, at least for this specific configuration. In fact, it may be sufficient to simply lengthen $\Delta t_{predictor}$ by a small amount while keeping the shorter duration constants the same. Such model-specific or experiment-specific fixes, however, are less than satisfactory. The main concern is not how to eliminate harmful rounding errors, but rather how to prevent them from going undetected. With the help of Figure 8 we can see that the 73.0% result is suspicious; it differs from configurations expected to yield statistically equal accuracies. Yet this verification procedure is not generally applicable, so temporal rounding errors may well go unnoticed.

The 73% configuration is an interesting case where the relevant time scales are just sufficiently disparate that floating-point arithmetic has a significant impact on the simulation results. As the time scales diverge, the simulation runs become nearly impossible to salvage with minor implementation improvements. Consider the configuration yielding 79.1% accuracy, just slightly above and to the right of 73% in Figure 8. The problems with this simulation run can be seen in the first of 100 predictions, shown in Figure 10. As before, the last measurement occurs after the end of the drift signal due to rounding error. But here the noise signal ends extremely early, with all but a few segments rounding to zero. The reduction in noise promotes consistency in the low-frequency measurements, which should bias the accuracy upward. A third problem is that the durations

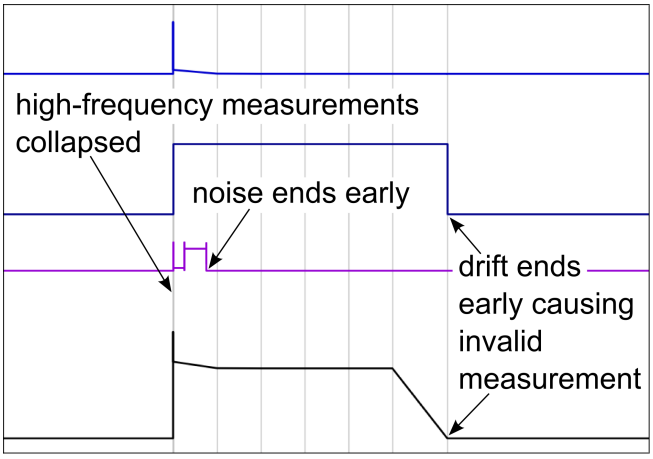


Figure 10. Impact of floating-point time values at $\Delta\bar{t}_{occurrence} = 10^{7}\,\text{s}$ and $\Delta\bar{t}_{incident} = 10^{-7}\,\text{s}$. The drift signal ends just slightly ahead of the last measurement. Also, a variety of other problems are evident.

between the high-frequency measurements have rounded to zero, and so the first 6 measurements now occur at a single point in simulated time. As we approach the top right corner of Figure 8, the scales become so disparate that all types of events seen in Figures 9 and 10 collapse to a single instant. This collapsing of events promotes consistency, causing prediction accuracy to be overestimated.

Revisiting Figure 8, observe the three failures for configurations with $\Delta\bar{t}_{occurrence} \in \{10^{5}\,\text{s}, 10^{6}\,\text{s}, 10^{7}\,\text{s}\}$ and $\Delta\bar{t}_{incident} = 10^{-8}\,\text{s}$. With these particular combinations of time scales, the $k_{incident}$ message is occasionally sent before the $y_{feature}$ needed to predict the class of the incident. The model has a condition to fail in such cases. The condition could be removed, allowing the simulations to complete, but the results would be invalid.

To summarize the experiment, a number of the drawbacks listed in Sections 3.1 and 3.2 are evident in the results produced with 32-bit fixed-point, 64-bit fixed-point, and 64-bit floating-point time representations. The experiment excludes 32-bit floating-point time values, but with only 24 bits of precision the outcome would be poor for most if not all of the 100 configurations. A clear trend in all the results is that the impact of temporal rounding error increases with the discrepancy between a model's longer and shorter time scales. In the fixed-point case, the longest time scale compels the selection of a long time quantum, and problems arise when this quantum approaches or surpasses the shortest time scale. Although there is no time quantum in the floating-point case, rounding errors proportional to the longest time scale cause varying degrees of havoc as they approach the shorter time scales.

3.4. Time implementation strategies

Various strategies have been used to implement time representations in the form of data types and associated operations. While a number of these strategies mitigate some of the drawbacks of fixed- and floating-point time values, no existing implementation seems to satisfy all concerns pertaining to the collaborative development of multiscale simulation models.

The first strategy we consider could be described as a "brute force approach" to representing simulated time. The idea is to use a fixed-point time representation consisting of (a) a universal time precision fine enough for all intended users, and (b) an arbitrary-precision integer multiplier. Arbitrary-precision integers are not constrained to 32 or 64 bits, but rather expand in memory as needed to accommodate increases in magnitude without sacrificing resolution. This representation addresses both the range limitations of 32- and 64-bit fixed-point time values as well as the undesirable rounding effects associated with floating-point time values.

To illustrate the advantages of a brute force approach to time representation, we extend the experiment in Section 3.3 to include fixed-point time values with arbitrary-precision multipliers. We select a time quantum of 10^{-30} seconds, which appears to be sufficiently short for any computer simulation we have encountered in the literature. As shown in Figure 11, believable mean accuracy values close to 76% are reported for all combinations of time scales.

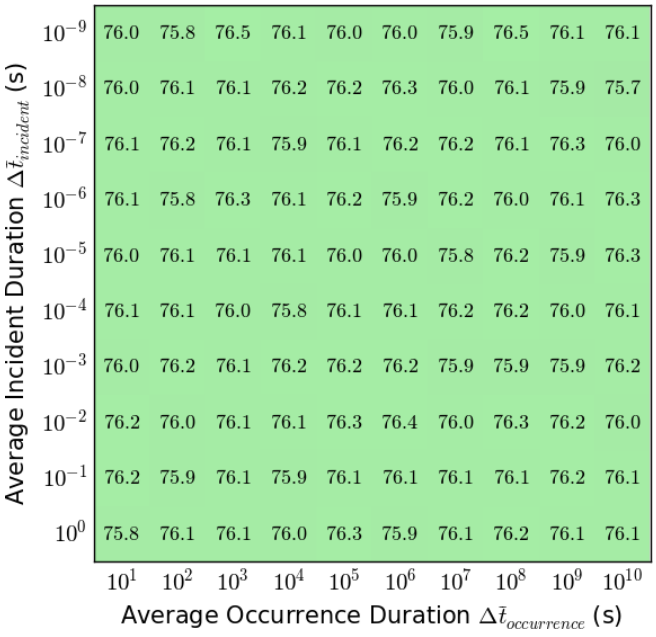


Figure 11. Mean accuracy results for Section 3.3 experiment using arbitrary-precision fixed-point time values.

Though it seems to eliminate all of the many problems encountered in the Section 3.3 experiment, a brute force

approach to time representation has two drawbacks. The obvious drawback is a reduction in computational efficiency when compared with standard 32- or 64-bit alternatives. But our main concern with the brute force approach is its reliance on a common time precision. Modelers might assume different precision levels, causing software testing/maintenance issues. Moreover, no universal time precision is suitable for all disciplines.

Consider the 10^{-30} s time quantum used to produce the Figure 11 results. We would expect quantum chromodynamics researchers to be satisfied with this precision level. Molecular biologists might find it unnecessarily fine-grained, yet tolerable. Modelers concerned with human-scale systems would find 10^{-30} s absurdly short, and might decide against adopting the time representation. Computational astrophysicists would find it even less acceptable. While the underlying issue is the perception of performance loss, the important point is the discriminating nature of the approach. The astrophysicist would not want to dedicate over 150 bits of precision to every time value knowing that experts of small-scale phenomena need dedicate only 64 bits. Choosing a coarser precision would better accommodate the modelers of large-scale systems, but might exclude modelers of small-scale systems.

In short, the brute force approach retains the fixed-point representation's inconvenient trait of requiring a common time precision.

Another implementation strategy involves the use of a rational number data type, which stores both a numerator and denominator as separate integers. Rational simulated time values are explored by Vicino et al. [2] in conjunction with a hybrid representation that strives to improve computational efficiency with special treatment for powers of two. To simplify the discussion, we focus on basic rational number data types without such optimizations. We assume that the numerator and denominator are both arbitrary-precision integers. Rational numbers are compelling in that they accommodate exact results not only for addition and subtraction, but also for multiplication and division. Rational time values require no preselected precision levels, have no range limitations, and can exactly represent durations such as $\frac{1}{3}$ s that involve neither binary nor decimal fractions.

While rational time values appear to solve a number of rounding problems, their benefits over fixed-point representations may be limited in practical situations. For example, consider a model that generates messages separated by exponentially distributed durations of simulated time. Suppose the time constant is 17 s. Theoretically, these durations are irrational numbers, but in practice they will be rounded off. If time values are rational numbers, one must arbitrarily select a precision level, and the most obvious approach is to fix the denominator. Suppose the denominator is fixed at 1,000,000. The numerator is then exponentially distributed with a mean value of 17,000,000. Although rational numbers do no harm to the result here, they offer little advantage over a fixed-point representation with a time quantum of $1\,\mu$s. Some modelers may prefer the fact that with rational numbers, the precision is implicit in the rounding that occurs within each model. Other modelers may prefer precision levels to be explicitly specified.

The C++11/14 Chrono Library provides a fixed-point representation in which the multiplier is an integer and the precision level is a rational number [52]. Although the number of bits of each time value is not strictly standardized, it appears typical that the multiplier and the numerator/denominator of the precision level are limited to 64 bits for the sake of efficiency. The precision level is a template argument and must therefore be inferrable at compile time. This promotes efficiency and early error detection, though for certain applications it might be preferable to determine the precision at run time. A reliance on templates discouraged us from adopting the Chrono Library as a basis for our multiscale time representation, but we still consider it a viable option for fixed-point simulated time values in C++.

One final implementation strategy is the use of a 64-bit floating-point number as the multiplier in a fixed-point time representation. The key observation is that a 64-bit floating-point number can exactly represent all consecutive positive integers with a magnitude of at most 2^{53}, or 9,007,199,254,740,992. Using an object-oriented programming language such as C++, a floating-point multiplier and a precision level can be encapsulated in a class. When an arithmetic operation is invoked on an instance of that class, the calculation is performed first with floating-point arithmetic, and the resulting multiplier is rounded if needed. An important caveat is that the multiplier not exceed $2^{53} - 1$.

According to the definitions in Sections 3.1 and 3.2, the strategy described above constitutes a fixed-point representation notwithstanding the internal use of a floating-point number. An encapsulated floating-point multiplier reduces the need for type conversions when performing time value operations involving multiplication and division. Furthermore, IEEE 754 floating-point numbers include a representation of infinity and its associated mathematical rules, providing a convenient way to incorporate infinite durations.

4. Multiscale time representation

Having examined both fixed-point and floating-point time values as well as a number of implementation strategies, we remain without a computer representation of simulated time that convincingly addresses the needs of collaborative, multiscale modeling and simulation. Particularly unsettling are the potential consequences of

floating-point time values. It is true that the most harmful effects occur only with vastly disparate time scales. But the subtlety of these effects, their tendency to surface in a variety of forms, and the fact they occur without warning leads us away from the floating-point option. Our solution more closely aligns with the fixed-point approach in that time durations associated with models include precision levels and multipliers. Yet we avoid introducing any practical range limitations, as well as any need to impose a common time quantum on the entire simulation. Despite the need for arbitrary-precision data types in certain places, the solution is itself a multiscale approach in that its associated event scheduling and recording algorithms can be implemented using mostly 64-bit operations. Here we describe the principles, key features, and design of the proposed multiscale time representation.

4.1. Representation principles

The proposed representation of simulated time is based on the following principles:

1. Every atomic model should have a specified time precision, and modelers should be able to assume that all time durations encountered by any instance of the model are rounded to this precision level. There is no universal time quantum that all modelers must agree on, as inevitable disagreements would then discourage the sharing of models. Furthermore, there is no selection of a common precision level on a per-simulation basis, as this might contradict the levels specified for one or more models included in the run. Respecting a model's specified time precision promotes consistent model behavior.

2. An atomic model's precision level is assigned using a base-1000 SI time unit (i.e. kiloseconds, seconds, milliseconds, etc.). Although minutes and hours may not serve as specified precision levels, we note that multiples of these durations can be exactly represented with a 1 s time precision. The restriction of time quanta to base-1000 SI units means that quantities such as $\frac{1}{3}$ s or fm/c cannot be exactly represented. We consider this limitation outweighed by the advantage of having a common factor (i.e. 1000) separate each assignable time unit (e.g. nanoseconds) from the next (e.g. picoseconds).

3. Atomic model code should use 64-bit fixed-point time values. The rationale for a single representation is to simplify the reading and writing of model-specific code for the benefit of modelers, who may be experts in a variety of disciplines but not necessarily software engineering. The restriction to 64 bits ensures that most mathematical operations on time values will be supported on nearly any platform, and that they will be computationally efficient. The fixed-point option avoids any unexpected behavior arising from floating-point arithmetic.

4. Whereas atomic models should be restricted to 64-bit fixed-point time values, a simulator may be implemented using an assortment of time-related data types and data structures. The rationale for tolerating greater complexity in simulator code is that its developers are expected to have considerable software engineering expertise. The internal complexity of a model-independent simulator must be hidden from the domain experts who apply it to their models.

5. An atomic model instance's simulated time values may be approximate, so long as any error is bounded by the model's time quantum δt. Uncertainty in the time durations which separate events is a necessary consequence of the principles above, but this uncertainty should be bounded.

6. Whereas time values encountered by atomic models may only approximate the actual progression of simulated time, a simulator must maintain the exact current time point and the exact time points associated with recorded and scheduled events.

7. Although simulators may use arbitrary-precision arithmetic in places to satisfy the principles above, 64-bit operations should be used where possible to reduce memory and processing requirements.

4.2. Representation features

The principles listed in the previous section demand a solution that maintains the convenience and predictable rounding behavior of a fixed-point representation, yet tracks event times with a degree of exactness traditionally accomplished only through the exclusive use of arbitrary-precision data types. The features described here collectively satisfy these requirements. The primary novelties of our approach include a scale-related rationale for distinguishing between durations and time points (Section 4.2.1), the notion of "perceived time" in the context of scale (Section 4.2.2), the operation called "multiscale time advancement" (Section 4.2.3), the incorporation of epochs into the event scheduling mechanism (Section 4.2.4), and the re-purposing of the event scheduling data structure for tracking elapsed durations (Section 4.2.5).

4.2.1. Durations vs. time points

Certain computer representations of time use the same data type for all time values, whereas others use different data types for durations of time and points in time.

In the C++11/14 Chrono Library, durations and time points are distinguished so that duration-related code can be more easily reused among systems that differ in the epochs used to encode dates[52]. An additional benefit of this approach is that certain operations can be defined for durations only, or time points only, but not both. For example, it makes sense to multiply a 20-second duration by 3 to obtain a 1-minute duration, but multiplying the time point March 14, 1:59 AM by 3 requires a reference point, which is generally arbitrary.

For the multiscale time representation, we must distinguish between durations and time points simply because Principle 3 of Section 4.1 demands 64-bit time values for model implementations, whereas Principle 6 requires simulators to use arbitrary-precision time values to track event times. Event times are time points, so our proposed time point data type has a variable number of digits. In model code, we discourage the use of these arbitrary-precision time values by avoiding the need to express event times. This is done by adopting a core convention of the DEVS formalism whereby models express time values in durations relative to the current point in simulated time. We refer to these time values as *elapsed durations*, measured from the most recent past event, and *planned durations*, measured to the most imminent future event of those that are currently planned. For example, if the simulator labels a previous, current, and future event with the time points 58, 67, and 72, respectively, the model might only be aware that the elapsed duration is 9 and the planned duration is 5. Our representation therefore complements the arbitrary-precision time point data type with a duration data type based on 64-bit operations. The duration type is used to represent elapsed durations, planned durations, and other relative quantities of time.

Listed below are several key operations involving duration operands (Δt), time point operands (t), or operands of both data types (Δt, t):

- **Negation** (Δt): The unary operation $-\Delta t$ reverses the sign (positive/negative) of the duration.
- **Scaling** (Δt): The binary operations $a{\cdot}\Delta t$ and $\Delta t{\cdot}a$ multiply the magnitude of the duration by the real number a.
- **Binary addition** (Δt): The binary operation $\Delta t_{\mathsf{A}} + \Delta t_{\mathsf{B}}$ adds the magnitudes of two durations to yield a new duration.
- **Binary subtraction** (Δt, t): The binary operation $\Delta t_{\mathsf{B}} - \Delta t_{\mathsf{A}}$ subtracts the magnitudes of two durations to yield a new duration, whereas the binary operation $t_{\mathsf{B}} - t_{\mathsf{A}}$ subtracts two time points to yield a duration. The resulting durations are always exact provided the difference between the operands can be represented using the 64-bit representation; if the result requires too many bits, the operation yields an infinite duration.
- **Gap estimation** (t): The binary operation $t_{\mathsf{B}} \ominus t_{\mathsf{A}}$ yields a duration that approximates the difference between the two time points. The approximation must be sufficiently accurate that the earlier time point t_{A} eventually reaches the later time point t_{B} if t_{A} is repeatedly advanced by $t_{\mathsf{B}} \ominus t_{\mathsf{A}}$.
- **Time accumulation** (Δt, t): The operation $t + \Delta t$ adjusts the time point by the exact value of the duration.
- **Time advancement** (Δt, t): The operation $t \triangleright \Delta t$ adjusts a time point representing the current time according to an operation that may not give the same result as time accumulation ($t + \Delta t$). A specific form of this operation called "multiscale time advancement" is introduced in Section 4.2.3 to honor the specified precision of each integrated model.

We will revisit some of these operations as we discuss other features of the multiscale representation and its design. The mathematics associated with these and other operations can be found in Appendix A.

4.2.2. Perceived time

Principle 5 of Section 4.1 states that an atomic model instance's simulated time values may be approximate, so long as any error is bounded by the model's time quantum δt. This scale-related form of approximation is a necessary consequence of Principles 1–4, and here we elaborate on the concept. In essence, an atomic instance can not necessarily determine the current simulation time t. From the instance's perspective, t is anywhere in the range $\hat{t} \le t < \hat{t} + \delta t$, where δt is the model's time quantum and $\hat{t}$ is its *perceived time*. In other words, the perceived time is the actual time rounded down to a multiple of the model's time quantum.

As illustrated in Figure 12, different perceived times coexist within a multiscale model. In the scenario shown, every model with a forest-unit time precision has a perceived time of 375. This means that the current time t is in the range $375 \le t < 500$. Models with a tree-unit precision perceive the current time as 425, though based on their knowledge the current time could be almost as advanced as 450. Leaf-unit precision models perceive the current time correctly as 448, assuming they are the finest-scale models in the system. Yet as far as the models are aware, the current time is in the range $448 \le t < 449$. Note that the factor of 5 separating one time precision from the next is a convention we use only for illustrative purposes. In the actual representation, successive precision levels are separated by factors of 1000, consistent with Principle 2.

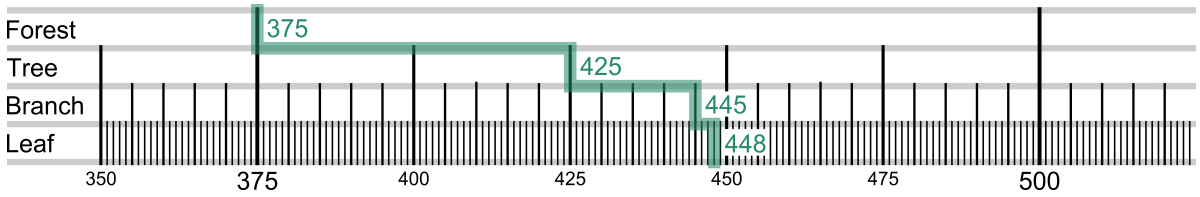


Figure 12. A current time of 448 at leaf-unit precision is perceived as 445, 425, and 375 at the branch-, tree-, and forest-unit precision levels.

Perceived time allows durations to be accumulated error-free at every independent scale. Consider, for example, a model with 1 s precision and 1000 s time steps. The simulation begins at $t = 0$, and an instance of the model schedules an event at $t = 1000$. But suppose the instance receives a message at $t = 2.2$. To maintain its scheduled $t = 1000$ event, the instance must then yield a planned duration of 997.8 s. Inconveniently, its 1 s precision level only allows it to produce a planned duration of 998 s. If conventional time accumulation is used, the future event will be scheduled at $t = 1000.2$, introducing error. Such errors can then accumulate over time. But if the 998 s planned duration is applied to the perceived message time ($t = 2$ instead of $t = 2.2$), then the $t = 1000$ event occurs exactly when planned.

Perceived time points are not explicitly represented in any of the data types or data structures in our approach. They exist only implicitly as a result of an operation called multiscale time advancement, explained in Section 4.2.3. However, an expert modeler can make perceived time explicit if he/she so choses. Let us assume a simulation begins at $t = 0$. An instance of the time point data type can be stored as a state variable in an atomic model instance, and set to $t = 0$ on initialization. If the time point is then advanced—using multiscale time advancement—by every encountered elapsed duration, the time point will track the perceived time. If this is done by the smallest scale model in the system, the associated time point is the current time. Again, we consider this an expert technique. Domain experts need not use the time point data type. In fact, they need not be familiar with the concept of perceived time. It is generally sufficient to know that some models measure time more precisely than others, and that the simulator sorts out any discrepancies.

4.2.3. Multiscale time advancement

Multiscale time advancement is the operation that introduces perceived time into a simulation, despite the fact perceived time values are not explicitly represented. Essentially, multiscale time advancement increases a perceived time point by exactly the duration specified, even though the actual time point may increase by a shorter duration. More precisely, the actual time point is increased by the shortest possible amount such that the model specifying the duration still perceives time as if it progressed exactly as much as expected.

Figure 13 illustrates multiscale time advancement in three distinct scenarios. In all three cases, a time point of 448 leaf units is advanced by a duration with a magnitude of 100 leaf units. However, the duration is expressed with a different precision level in each scenario, which changes the outcome of the operation. In the upper plot, the duration is expressed as 100 leaf units, and time progresses by the expected amount. In the middle plot, the perceived time of 445 at the branch-unit scale increases by exactly the specified duration of 20 branch units, but the actual time increases by only 97 leaf units (from 448 to 545). In the lower plot, the perceived time of 425 at the tree-unit scale advances by exactly 4 tree units, but the actual time increases by only 77 leaf units (448 to 525).

Section 4.2.1 introduced the notation for multiscale time advancement, $t \triangleright \Delta t$. The notation is inspired by Nutaro [53], who uses the $\triangleright$ operator for advancing a rather different form of time representation. Nutaro's time values are pairs (t, c), where t is the simulated time and c is a count which orders events that share a simulated time point. Extensive work has been done on similar time representations incorporating one or more non-physical components [54,55,56,57,58,59,60,61]. The multiscale representation described here deals only with simulated time: the t component in Nutaro's (t, c) time values. Yet the manner in which multiscale time advancement truncates small-scale information is somewhat analogous to the reseting of the c component, which occurs when a (t, c) time point is advanced by a positive duration of simulated time.

4.2.4. Event scheduling epochs

As the multiscale representation was under development, a primary consideration was the question of whether the proposed time values and operations would reasonably accommodate the complete set of data structures and algorithms needed to implement a generic simulator. As the focus was on discrete-event simulation, we dedicated much of our efforts to the enhancement of priority queues that handle the scheduling of future events.

Inconveniently, neither the time point data type nor the duration data type are suitable for tracking the tim-

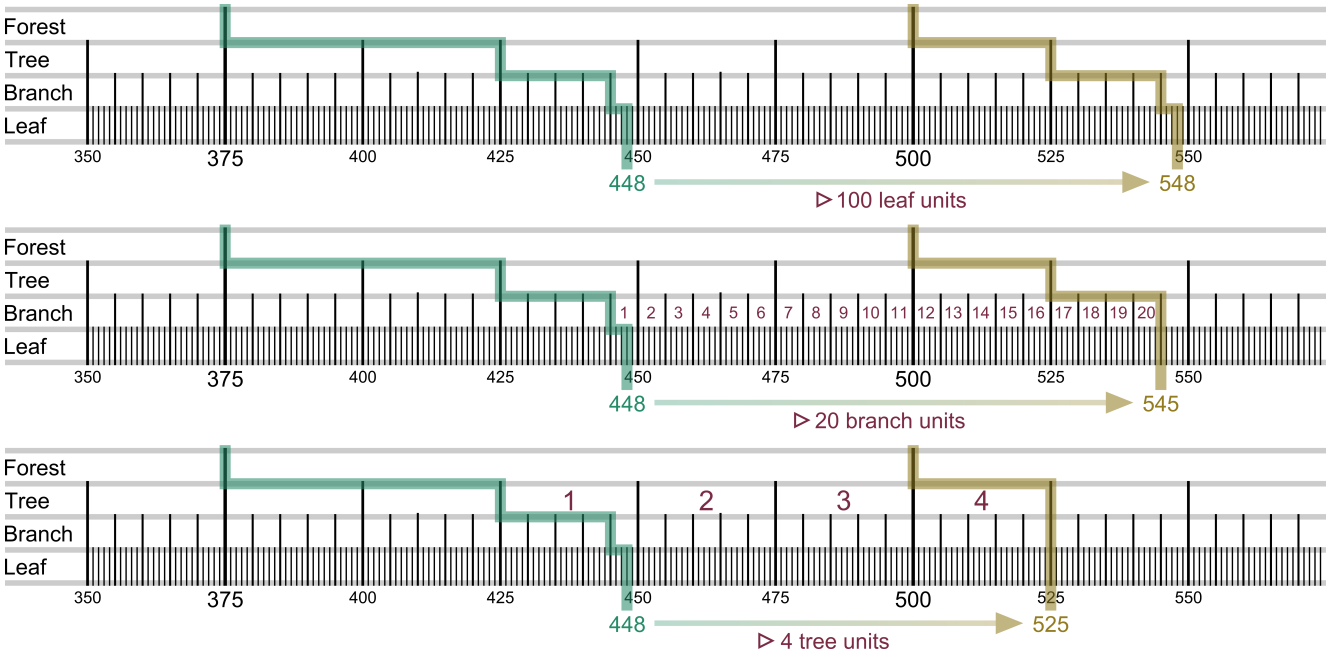


Figure 13. Using multiscale time advancement, advancing a time point (448) by equal durations (100 leaf units = 20 branch units = 4 tree units) produces a different result depending on the time unit of the advancement duration.

ing and ordering of large numbers of future events. Assigning an arbitrary-precision time point to every future event would contradict Principle 7 of Section 4.1, which states that 64-bit operations be used where possible to reduce memory and processing requirements. And although our duration time values are based on 64-bit operations, they are inappropriate for a different reason. Recall from Section 4.2.1 that planned durations are measured from the current point in simulated time. If every future event was tracked using its planned duration, then all of the durations stored in the queue would have to be updated whenever the current time advances. We prefer to avoid such obvious inefficiencies.

Note that the issue of how to represent future event times is orthogonal to the choice of priority queue algorithm. This point deserves elaboration. Rönngren and Ayani [62] compare the performance of several priority queue algorithms including the implicit binary heap, the splay tree, and the calendar queue. Any of these methods could be combined with the multiscale time representation we propose. What our approach affects is the encoding and interpretation of the time value associated with each event. To distinguish between the challenge of how to physically handle future event information, and the challenge of how to encode and interpret event times for multiscale simulation, we adopt a new term. In the context of our approach, a time queue is an event-tracking data structure that encapsulates both a priority queue algorithm (i.e. one of those compared by Rönngren and Ayani [62]) and our method for processing time values associated with different scales.

Our time queue data structure introduces epochs into the event scheduling process. We define an *epoch* as the time period starting at an *epoch reference time*, or "epoch date", and ending at the most distant time point that can be represented as a positive fixed-point offset from the reference time. For example, the January 1, 1970 epoch date popularized by C, C++, and Unix is widely used in conjunction with a 1s precision level and a signed 32-bit integer multiplier. This convention gives rise to the so-called "Year 2038 Problem", the prospect of widespread system failures coinciding with the end of the epoch. Incidentally, this is further evidence of the inadequacy of 32-bit time values.

The epochs we introduce are juxtaposed end on end, allowing the full extent of a simulation run to be accommodated regardless of how much simulated time elapses or how precisely event times are resolved. Every future event time is then stored as an offset from the beginning of its encompassing epoch. We refer to the stored offsets as *planned phases*. Planned phases, planned durations, and epochs are illustrated in Figure 14. Note that all time points and durations in this diagram are specific to the leaf scale, regardless of where they are drawn. In the example, an epoch at the leaf scale coincides with a single time quantum at the forest scale.

The *current epoch* contains the current time. In Figure 14, the current time is 422, and the current epoch stretches from time 375 to 499. An event scheduled for time 479 is also in this epoch. The planned duration for this event is 57 $(479 - 422)$, but will decrease as time advances. The corresponding planned phase will retain

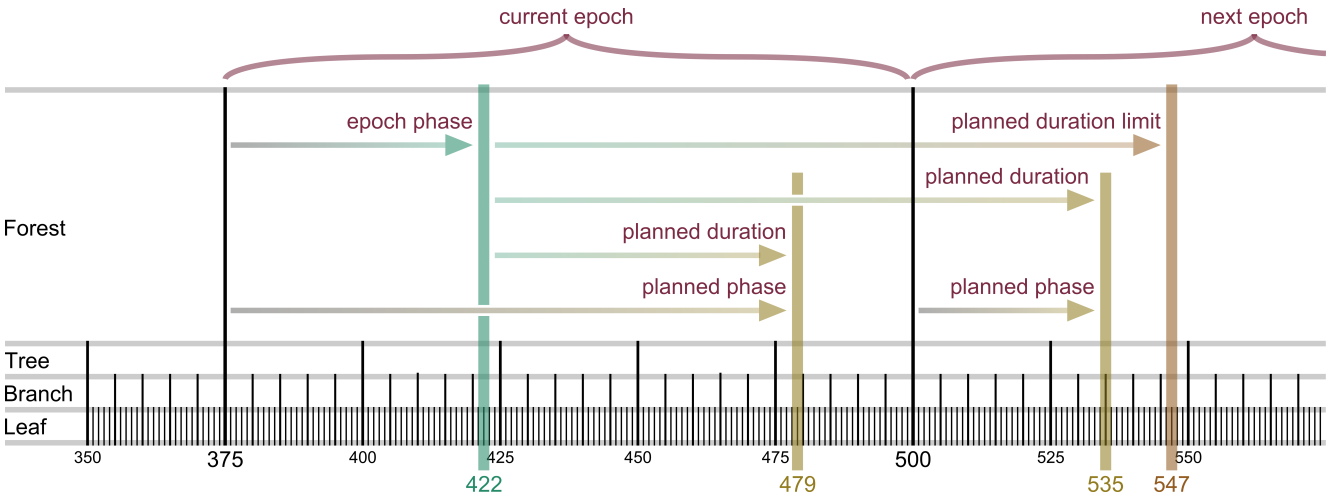


Figure 14. The illustrated time queue tracks two planned durations, which are measured relative to the current time (422), by storing the corresponding planned phases, which are measured relative to the beginnings of epochs. The first planned duration is within the current epoch (375-499), which contains the current time. The second planned duration is within the next epoch (500-624), which will eventually become the current epoch. All time points and durations shown are specific to the leaf scale. Epochs at the leaf scale coincide with time quanta at the forest scale.

its value of 104 (479 − 375) as time advances, which is why the time queue data structures stores planned phases internally. Planned phases are converted to and from planned durations as needed.

The *next epoch* begins after the current epoch ends. In Figure 14, the next epoch starts at time 500 and contains a scheduled event at time 535. Again, the planned phase is stored because it remains constant as time advances. Observe that a planned phase in the current epoch is at least as long as its corresponding planned duration; in the next epoch, a phase is always shorter than its planned duration because it is measured from a more advanced point in time (i.e. time 500 instead of 375).

A compelling feature of this approach is that it is unnecessary to record which epoch contains each planned phase. Given only an event's phase value, one can determine its associated epoch, its planned duration, and its time point. The calculation requires the current time t, the only time point stored by a time queue. Using t, one computes the *epoch phase*, the number of time quanta separating t from the beginning of the current epoch. If a planned phase is at least the epoch phase, the event is in the current epoch. If the planned phase is less than the epoch phase, the event is in the next epoch. Note that these are the only two cases. Scheduled events can never precede the current epoch because they would have already occurred. Likewise, the epoch that begins after the next epoch is irrelevant, as we impose a limit of one epoch width on the duration data type used to specify planned durations. In Figure 14, the epoch width and planned duration limit are both 5^3, or 125. The actual representation uses an epoch width and duration limit of 1000^5, As explained in Section 4.3.

Figure 14 shows only events scheduled by models with a common time precision (leaf-unit precision, in this case). A number of complications arise when tracking events scheduled by models with different precision levels. Yet even if all events are handled by the same time queue instance, it remains possible to store future event times using planned phases, and to compare these phases without resorting to arbitrary-precision arithmetic. The algorithms required to achieve this functionality are provided in Appendix A.5.

4.2.5. Other event tracking features

In addition to tracking future events, it is often necessary to store and retrieve durations and time points associated with past events.

To record all past event times as they occur, we define a relatively simple data structure called the time sequence. When an instance of this data structure receives its first event time as an instance of the time point data type, it stores the full arbitrary-precision value along with a zero-valued offset of type duration. When new event times are added, the structure appends additional duration-valued offsets from the initial time point. If the new event time is sufficiently advanced or sufficiently precise that it cannot be exactly represented with an offset, then the full time point is recorded along with a duration of zero. Thereafter, additional event times are measured from this new time point. Details can be found in Appendix A.4.

Finally, though discrete-event simulation is characterized by the use of a priority queue, theory from Zeigler et al.[3] reveals that one additional data structure is needed to support all applications. A central concept

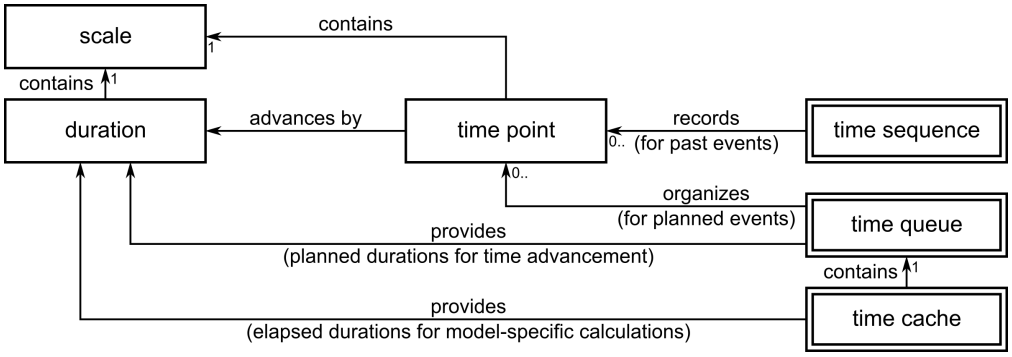


Figure 15. Multiscale time representation data types (single border), data structures (double border), and relationships (arrows).

underlying the DEVS formalism is the observation that certain state transitions depend on the elapsed duration, the time elapsed since the previous event. We therefore define a time cache data structure responsible for providing elapsed durations by tracking the time points of the previous event for each model instance. This is different from the time sequence, which can record all event times and is not optimized for the previous.

Conveniently, the time cache data structure can be designed with ease using the following technique. Instead of attempting to track a previous event as it recedes into the past, the idea is to track an imaginary future event as it becomes progressively more imminent. The previous event and its imaginary future event are always separated by the maximum representable duration, $1000^5 - 1$ time quanta, so tracking one is as effective as tracking the other. We track the imaginary future event so that we may simply re-purpose the time queue data structure. Thus a time cache encapsulates a time queue; information about past events is maintained by internally tracking imaginary future events. More information is provided in Appendix A.6.

4.3. Representation design

We propose that the features described in Section 4.2 be organized in a computer representation of simulated time consisting of three data types named scale, duration, time point, as well as three data structures named time sequence, time queue, and time cache. All six elements and their relationships are shown in Figure 15. The separation of durations and time points was discussed in Section 4.2.1. The reason we abstract the concept of scale into its own data type is that it can be reused for both the duration and time point data types, as well as distance-related data types in possible future work toward a multiscale representation of space. As previously discussed in Sections 4.2.4 and 4.2.5, the time sequence, time queue, and time cache data structure collectively support the recording of past events and the scheduling of future events in a discrete-event simulation.

Two mathematical constants, β and η, play an important role in the representation's design. The *base constant* β is a factor that separates one allowable time unit from the next. Since the time unit representing a model's precision is roughly proportional to the model's time scale, we could state that β also separates one scale from the next. To adhere to Principle 2, β must be 1000, though other base constants are possible in theory.

The *epoch constant* η establishes β^η as the limiting multiplier of the duration data type. We use $\eta = 5$ for the following reason. Recall from Section 3.4 an implementation strategy in which an integer-valued multiplier is represented using an encapsulated 64-bit floating-point number. We adopt this technique, and must therefore choose a limiting multiplier less than 2^{53}, or 9,007,199,254,740,992. Yet as explained below, we also require that this limit be a power of β. The largest value of 1000^η less than 2^{53} is one quadrillion, or 1000^5. Hence $\eta = 5$, and the duration data type's fixed-point multiplier m must satisfy $10^{-15} < m < 10^{15}$.

The decision to use a 10^{15} multiplier limit instead of 2^{53} is to align a full epoch at one scale with single time quantum at a larger scale. The alignment of an epoch with a time quantum occurs when there are exactly η steps between the two scales. For example, with $\eta = 5$, an epoch at a scale with an attosecond precision level (1000^{-6} s) is exactly one millisecond (1000^{-1} s). The relationship between time quanta and epochs was illustrated in Figure 14 with the alternative set of constant values $\beta = 5$ and $\eta = 3$. Note that it would be impossible to render a full epoch's worth of quanta using the actual $\beta = 1000$, $\eta = 5$ conventions.

To help make the proposed representation reproducible, a detailed mathematical description of each data type and data structure is provided in Appendix A. The appendix uses a unique set of notations in order to distinguish the proposed multiscale time value operations from conventional arithmetic. For example, the scale data type represents a power of 1000, specifically 1000^{level} for some integer *level*, but we use the notation below in order to disallow unnecessary scale-valued operations such

as the multiplication of two scale values.

$$\blacklozenge^{level} \qquad \{\text{scale } (1000^{level})\}$$

The notation for duration values uses a clock symbol and clearly shows the multiplier m and precision level. The precision is represented using the scale data type.

$$m\blacklozenge^{u} \qquad \{\text{duration } (m \cdot 1000^{u}\,\text{s})\}$$

The time point notation uses a different clock symbol and also incorporates a scale-valued precision level. Instead of a single multiplier variable, this data type features a sequence of n digits d_i, where $0 \le d_i < 1000$.

$$[\underset{n-1}{d_{n-1}}, \ldots, \underset{2}{d_2}, \underset{1}{d_1}, \underset{0}{d_0}]\blacklozenge^{u} \qquad \{\text{time point}\}$$

In addition to describing these data types and their associated operations, Appendix A formalizes the key algorithms employed by the time sequence, time queue, and time cache data structures. Appendix B offers sample code and suggestions for implementing the multiscale time representation in C++.

5. Modeling and simulation using a multiscale time representation

A multiscale time representation has now been proposed, but how would it impact modeling and simulation? Here we answer this question. We focus first on issues relevant to modelers, particularly domain experts hoping to collaborate in the development of multi-domain, multi-paradigm, and multiscale simulations. We then present a prototype implementation and its application to the prediction system model of Section 3.3.

5.1. Implications for multiscale modeling

Modelers are experts in their domains who may have considerable coding experience, but generally lack a systems engineer's familiarity with computer technology and software engineering practices. Ideally, modelers would enjoy the benefits of a multiscale time representation with minimal exposure to its complexity. For this reason, most modelers should use only the scale and duration data types. The time point data type and the time sequence, time queue, and time cache data structures are intended primarily for systems engineers who develop simulation frameworks and environments. Realistically, there may be occasions where an expert modeler uses one of these data structures within a model. But typically, a model will store and process temporal quantities using the duration data type.

Aside from a modeler's direct exposure to the duration data type, the proposed time representation introduces a number of considerations pertaining to the modeling process. In the following, we explain why these considerations arise and how modelers should respond.

5.1.1. Selection of time precision

As emphasized throughout the paper, our approach requires almost every atomic model to have an explicitly specified time precision. The one exception is that some atomic models produce only zero or infinite planned durations and have no need for elapsed durations. These models can be described as having no time scale. They need not be given any precision level.

Multiscale models are constructed by integrating atomic models with different precision levels. Interactions among these models are coordinated using multiscale time advancement, which truncates every increase in the current time based on the time precision of the atomic model scheduling a future event. It turns out this truncation has no effect following a *planned event*: an event scheduled by an instance for itself. A planned duration can only be shortened following an *unplanned event*: an event triggered by an incoming message. The effect of this truncation is most pronounced when time is advanced by a single quantum.

Figure 16 depicts a scenario in which the current time is 449 leaf units. There is a model at the tree-unit precision level, and an instance of this model is currently scheduling an event with a planned duration of 1 tree unit. A single tree unit equals 25 leaf units; hence, under a conventional time accumulation approach, the future event would be scheduled at a time of 474 leaf units (449 + 25). Using multiscale time advancement, however, the event time is truncated at the model's precision level. The event is therefore scheduled at time 450, only one leaf unit beyond the current time. (Note: the instance's perceived time increases from 425 to 450, but the current time advances from 449 to 450.) In effect, the planned duration suffers a 96% error since the event is scheduled after 1 leaf unit instead of 25.

In general, if a large-scale model produces planned durations Δt_p equal to one unit of its precision level, and if this model is influenced by smaller-scale models, then temporal rounding errors may approach 100% of Δt_p. Fortunately, this problem is easily addressed by choosing a time quantum considerably shorter than the model's planned durations. If Δt_p is increased from one quantum to only two quanta, the maximum error drops to just under 50%. As a rule of thumb, we recommend that a model's time precision be one millionth of its time scale, rounded down to a base-1000 SI unit. If this guideline is followed, the situation in Figure 16 will rarely occur.

5.1.2. Limit on planned durations

We turn our attention to the scheduling of future events, where the key constraint is that every finite planned duration must be less than 10^{15} time quanta. Thus, while a model's time precision should be sufficiently fine to keep rounding errors well below its scale, an excessively

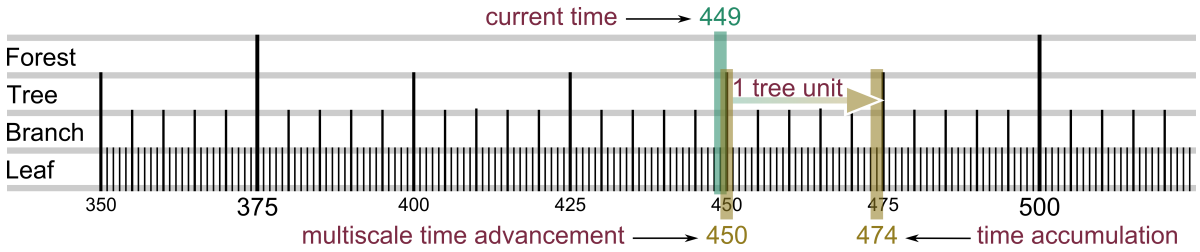


Figure 16. A scenario in which an event is scheduled with a planned duration of 1 tree unit, but the event time is truncated due to multiscale time advancement. The scheduled event occurs only 1 leaf unit into the future.

fine precision level will limit how far into the future one can schedule events. Let us observe that 10^{15} is equal to 10^6 times 10^3 times 10^6. Our rule of thumb attempts to keep most planned durations in the middle three orders of magnitude, at least 6 orders above the quantum and at least 6 orders below the 10^{15} limit. If a model's scale is roughly 5 s, one chooses a $1\,\mu$s precision and accepts $999,999,999.999999$ s (approximately 32 years) as the furthest duration into the future an event can be conveniently scheduled.

Although there is a 10^{15} quanta limit on the scheduling of future events, there is essentially no limit on the extent of a simulation run. For example, it is possible for a model to schedule a *preemptive event* every $5 \cdot (10^{14})$ time quanta, ensuring that the limiting duration of 10^{15} quanta never expires. With this technique and some bookkeeping code, the modeler can actually circumvent the limit and allow events to be scheduled arbitrarily far in the future. Yet such measures will rarely be necessary, since 10^{15} allows sufficient range and precision for most single-scale models.

5.1.3. Infinite elapsed durations

Arguably the most inconvenient aspect of the proposed time representation is the prospect of infinite elapsed durations. These occur when (a) an atomic instance enters a passive state by yielding a planned duration of ∞, (b) the instance remains undisturbed for at least 10^{15} time quanta according to its precision level, and (c) a message is then received triggering an unplanned event. Infinite elapsed durations are inconvenient because special cases may be required to handle them. Fortunately, if modelers forget to implement these special cases, there is good chance their simulations will fail in a noticeable way due to the tendency for ∞ values to propagate.

Modelers must understand that an infinite elapsed duration means the instance is in a steady state. If in fact the instance is not in a steady state, the modeler should not have allowed the instance to remain passive for 10^{15} or more time quanta. Our definition of *steady state* is a situation in which a response to an external stimulus no longer depends, in any predictable way, on exactly when the stimulus occurs. Hence when the actual elapsed duration overflows to ∞, no important information is lost in terms of how the model should behave. For example, if a state variable y approaches an asymptote $y_{\max}$, the modeler can approximate y as either $y_{\max}$ or $y_{\max} - \epsilon$ for some suitably small ϵ. On the other hand, if the steady state happens to be an oscillating pattern, one option is to choose a phase randomly. Imagine a wheel that is suddenly stopped after being allowed to spin for a very long time; its final rotation angle is more-or-less random. If needed, a modeler can avoid infinite elapsed durations altogether by scheduling preemptive events every $5 \cdot (10^{14})$ time quanta (see Section 5.1.2).

5.1.4. Extremely disparate scales

As previously emphasized, multiscale modeling is accomplished by integrating single-scale atomic models with different precision levels. There is effectively no bound on the diversity of scales that can be combined. Suppose one simulates the cosmic evolution of the Universe up to 10^{38} years when black holes theoretically become the only stellar-like objects. Suppose he/she then wishes to resolve molecular effects at representative locations in this dark, distant future. The overall simulation could be achieved with perhaps a yottasecond (10^{24} s) precision atomic model for the later stages of the Universe, a yoctosecond (10^{-24} s) precision atomic model for the molecular-level effects, and possibly terasecond, second, and picosecond atomic models to bridge the gap between these vastly disparate scales. It is unclear whether such a simulation will ever be undertaken, but comforting to know that even the most extreme multiscale modeling efforts need not be hindered by issues related to time representation. Models involving combinations of subatomic, molecular, biological, geological, and astrophysical time scales can be integrated with relative ease, and simulated using mostly 64-bit time value operations.

5.2. Prototype simulator and application

The DEVS formalism provides a basis for reusable simulators supporting a compositional style of model development [3]. Due to the generality of the formalism, a DEVS-based simulator can also serve as a foundation for multi-

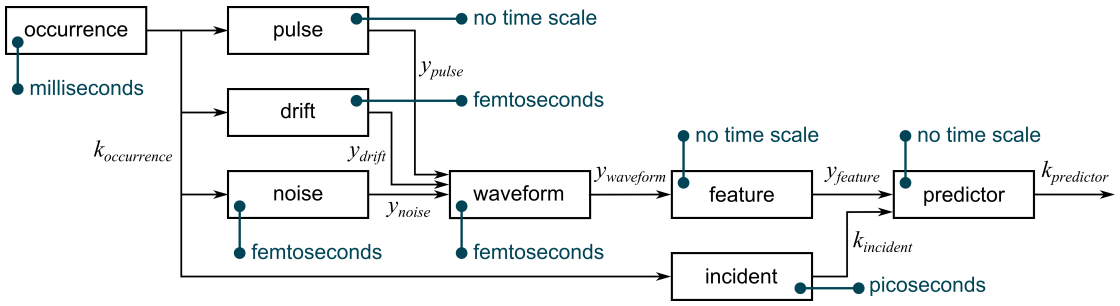


Figure 17. The prediction system model of Section 3.3 with time units specifying each model's precision levels.

paradigm modeling tools which accommodate a variety of general-purpose and domain-specific simulation techniques[63]. Our prototype simulator takes this highly regarded approach, and incorporates the proposed representation of simulated time. Although we make use of the existing convention that DEVS models express event times relative to the current time, every model must now be extended with a specified time precision. We test the prototype by applying it to the prediction system model of Section 3.3. As discussed below, the failures and extreme rounding errors observed when using fixed-point and floating-point time representations do not surface with the new simulator.

In reimplementing the prediction system model using the multiscale time representation, a precision level is specified for each of the eight atomic models. While we would generally want to use our rule of thumb (divide by a million, round down), this application is an unusual case where the two duration parameters $\Delta t_{occurrence}$ and $\Delta t_{incident}$ each vary over 10 orders of magnitude. The need to accommodate these vastly different time scales forces us to choose particular precision levels. In the case of the occurrence model, a minimum $\Delta t_{occurrence}$ of 10 s suggests a microsecond precision level based on the rule of thumb, but we choose milliseconds to accommodate the maximum $\Delta t_{occurrence}$ of 10^{10} s.

Figure 17 indicates the selected time precision for each atomic model in the prediction system. Models which react instantaneously in simulated time have no time scale, and thus no time precision. Observe that 27 orders of magnitude separate the longest simulation durations (i.e. the 10^{10} s maximum occurrence duration, multiplied by 10^2 occurrence-incident pairs) from the finest precision levels (i.e. femtoseconds, or 10^{-15} s).

The experimental results for all 100 combinations of $\Delta t_{occurrence}$ and $\Delta t_{incident}$ are shown in Figure 18. Unlike the fixed- and floating-point results in Section 3.3, the simulations neither fail nor deteriorate toward the top right corner where the scales become disparate. The results resemble those of the brute force approach in Section 3.4. But since each atomic model now has its own time precision, a wider range of disciplines can now be accommodated.

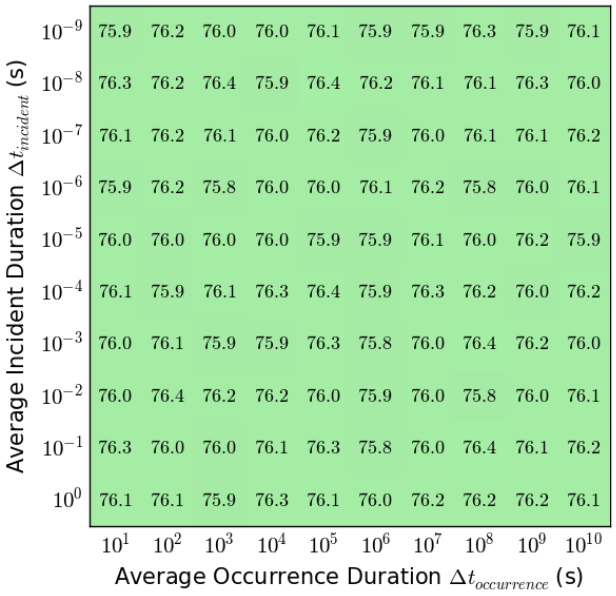


Figure 18. Mean accuracy results for the Section 3.3 experiment using the multiscale time representation.

Though the use of 64-bit operations wherever possible should promote efficiency, an analysis of computational speed and memory consumption remains as future work. Further experiments could help answer a number of performance-related questions. How does the use of a fixed-point, floating-point, brute force, or multiscale time representation affect run times for a representative sample of simulation models? Under what circumstances do time value operations become significant from an efficiency perspective in comparison with domain-specific computations? How can the proposed approach be optimized for time and/or space efficiency, and are such efforts worthwhile? Yet regardless of efficiency-related considerations, the multiscale time representation is justifiable as a scalable and domain-neutral strategy for controlling rounding error while avoiding the need for any universally accepted precision level. The approach promotes quality simulation results using conventions suitable for collaborative modeling efforts.

6. Conclusions

Scale, as defined in Section 2.1, pertains to the magnitudes of distances and durations that appreciably affect simulation results. It follows that rounding errors in the timing of events must be small relative to a model's time scale. But this requirement may be difficult to satisfy in a multiscale context if time is represented using standard 32- or 64-bit numbers, as rounding errors proportional to the longer time scales may distort behavioral patterns at the shorter time scales. In pursuit of general approaches for collaborative multiscale modeling, we have examined the implications of various representations of simulated time including the multiscale solution introduced in this paper. This work leads us to the following conclusions:

1. All types of 32-bit time values should be avoided in discrete-event simulation, except possibly in the most constrained applications. With 32 bits, only a narrow range of scales can be supported.

2. Binary floating-point time values are problematic and should generally be avoided. While floating-point is often considered the default representation for real numbers, its distinguishing characteristic is the fact that rounding errors are proportional to the magnitude of the represented value. This rounding behavior is often desirable, but not in the case of simulated time. As a simulation progresses, the current time variable increases, but there is rarely a reason that errors should increase as well. Errors associated with floating-point time values can have a severe impact in multiscale contexts.

3. Fixed-point time values with 64-bit integer multipliers could be considered a naïve approach to representing simulated time. One must choose a precision level based on the anticipated length of a simulation run, then hope that the shortest scales will be adequately resolved. The most ambitious multiscale applications may be difficult to accommodate.

4. Fixed-point time values with arbitrary-precision integer multipliers could be considered a brute force approach to time representation. In this case the shortest scales determine the precision level, and computing resources are acquired as needed to accommodate the full length of any simulation run. Multiscale efforts are supported, but arguably only within isolated disciplines. Collaboration between disciplines suffers because modelers are unlikely to agree on a universal time precision.

5. The solution presented in this paper constitutes a multiscale approach to the representation of simulated time. Precision levels specified on a per-model basis keep temporal rounding errors low relative to scale, which by definition minimizes the effects of these errors. Instead of relying on arbitrary-precision integers throughout the simulation code, 64-bit fixed-point time values are used within models and wherever possible in the simulator. Collaboration among disciplines is promoted, since models with disparate time scales can be integrated without the need to agree on any common time precision.

A recipient of the 2013 Nobel Prize in Chemistry awarded for multiscale modeling, Karplus[64] begins his lecture with a three-atom simulation developed in the 1960s. This serves as an example of how an early focus on single-scale simulations, combined with a willingness to collaborate, will ultimately give rise to multiscale considerations due to the inevitable differences in scale among models. Thus, if a simulation framework is intended to support the sharing and integration of models within a growing interdisciplinary community, one should fully expect multiscale modeling to become a priority even if it is not emphasized at the outset.

Because time representations are generally shared between models and simulators, they are difficult to replace once established in a framework. The fact that OMNeT++ switched to a fixed-point approach is a testament to the deficiencies of floating-point time values. Yet fixed-point representations are also limited if one's ultimate hope is to foster collaboration among disciplines. By incorporating a multiscale representation of simulated time such as the one presented in this paper, a framework's time value operations should never become a limiting factor as multiscale scenarios emerge.

General solutions that address the challenge of multiscale modeling will complement advances in multi-domain and multi-paradigm modeling, improving support for collaboration in systems science. As mentioned in Section 2.3, a new representation of time is one of two general solutions to multiscale challenges we find particularly compelling. The other is a multiscale representation of space. Most computer graphics libraries are based on floating-point spatial coordinates, and techniques such as normalization are used to cope with the resulting rounding errors. This approach will become untenable for models combining vastly different spatial scales. A multiscale alternative might include data types such as a unit vector with base-2 floating-point components (direction), a base-1000 floating-point spatial scalar (distance), a spatial vector with base-1000 fixed-point components (position), and an arbitrary-precision spatial vector to be used only where needed (reference point). In addition to effective representations of space and time, general support is needed for domain-neutral multiscale approaches such as those illustrated in Section 2.2. These challenges provide a way forward in addressing one of the most important bases for heterogeneity in the modeling and simulation of complex systems: a discrepancy in scale.

References

1. Hoekstra A, Chopard B, and Coveney P. Multiscale modelling and simulation: a position paper. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2021), 2014.

2. Vicino D, Dalle O, and Wainer G. A data type for discretized time representation in DEVS. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, 2014.

3. Zeigler BP, Praehofer H, and Kim TG. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, San Diego, CA, USA, second edition, 2000.

4. Gattass RR and Mazur E. Femtosecond laser micromachining in transparent materials. *Nature Photonics*, 2:219–225, 2008.

5. Ben-Nissan G and Sharon M. Capturing protein structural kinetics by mass spectrometry. *Chemical Society Reviews*, 40(7):3627–37, 2011.

6. KVA. Scientific background on the Nobel Prize in Chemistry 2013. Technical report, The Royal Swedish Academy of Sciences (KVA), 2013.

7. Castiglione F, Pappalardo F, Bianca C, Russo G, and Motta S. Modeling biology spanning different scales: An open challenge. *BioMed Research International*, 2014.

8. Bettini C, Dyreson CE, Evans WS, Snodgrass RT, and Wang XS. *Temporal Databases: Research and Practice*, chapter A glossary of time granularity concepts, pages 406–413. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

9. Bettini C, Wang XS, and Jajodia S. A general framework for time granularity and its application to temporal reasoning. *Annals of Mathematics and Artificial Intelligence*, 22(1):29–58, 1998.

10. Guo S, Hu X, and Wang X. On time granularity and event granularity in simulation service composition (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2012.

11. Goldstein R, Breslav S, and Khan A. A quantum of continuous simulated time. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2016.

12. Reynolds PF and Natrajan A. Consistency maintenance in multiresolution simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(3):368–392, 1997.

13. Davis PK and Bigelow JH. *Experiments in Multiresolution Modeling (MRM)*. Monograph Reports. RAND Corporation, 1998.

14. Heiner M and Gilbert D. Biomodel engineering for multiscale Systems Biology. *Progress in Biophysics and Molecular Biology*, 111(2–3):119–128, 2013.

15. Dada JO and Mendes P. Multi-scale modelling and simulation in systems biology. *Integrative Biology*, 3 (2):86–96, 2011.

16. Qu Z, Garfinkel A, Weiss JN, and Nivala M. Multiscale modeling in biology: How to bridge the gaps between scales? *Progress in Biophysics and Molecular Biololgy*, 107(1):21–31, 2011.

17. Elliott JA. Novel approaches to multiscale modelling in materials science. *International Materials Reviews*, 56(4):207–225, 2011.

18. Brandt A. Multiscale scientific computation: Review 2000. In Barth T, Chan T, and Haimes R, editors, *Multiscale and multiresolution methods*, volume 20 of *Lecture Notes in Computational Science and Engineering*, pages 3–95. Springer Berlin Heidelberg, 2001.

19. Weinan E. *Principles of Multiscale Modeling*. Cambridge University Press, Cambridge, UK, 2011.

20. Saunders MG and Voth GA. Coarse-graining methods for computational biology. *Annual Review of Biophysics*, 42(1):73–93, 2013.

21. Sarraf Shirazi A, Von Mammen S, and Jacob C. Abstraction of agent interaction processes: Towards large-scale multi-agent models. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(4):524–538, 2013.

22. Dzwinel W, Wcisło R, Yuen DA, and Miller S. PAM: particle automata in modeling of multiscale biological systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(3), 2016.

23. Abraham FF, Broughton JQ, Bernstein N, and Kaxiras E. Spanning the length scales in dynamic simulation. *Computers in Physics*, 12(6):538–546, 1998.

24. Brereton TJ, Kroese DP, Stenzel O, Schmidt V, and Baumeier B. Efficient simulation of charge transport in deep-trap media. In *Proceedings of the Winter Simulation Conference (WSC)*, 2012.

25. Nguyen VP, Stroeven M, and Sluys LJ. Multiscale continuous and discontinous modeling of heterogeneous materials: A review on recent developments. *Journal of Multiscale Modelling*, 03(04):229–270, 2011.

26. Kevrekidis IG, Gear CW, and Hummer G. Equation-free: The computer-aided analysis of complex multiscale systems. *AIChE Journal*, 50(7): 1346–1355, 2004.

27. Weinan E, Engquist B, Li X, Ren W, and Vanden-Eijnden E. Heterogeneous multiscale methods: A review. *Communications in Computational Physics*, 2(3):367–450, 2007.

28. Weinan E, Ren W, and Vanden-Eijnden E. A general strategy for designing seamless multiscale methods. *Journal of Computational Physics*, 228(15): 5437–5453, 2009.

29. Lockerby DA, Patronis A, Borg MK, and Reese JM. Asynchronous coupling of hybrid models for efficient simulation of multiscale systems. *Journal of Computational Physics*, 284:261–272, 2015.

30. Fritzson P and Engelson V. Modelica — a unified object-oriented language for system modeling and simulation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

31. Vangheluwe HLM, de Lara J, and Mosterman PJ. An introduction to multiparadigm modelling and simulation. In *Proceedings of the Conference on AI, Simulation and Planning in High Automony Systems (AIS)*, 2000.

32. Groen D, Zasada SJ, and Coveney PV. Survey of multiscale and multiphysics applications and communities. *Computing in Science Engineering*, 16(2): 34–43, 2014.

33. van Elteren A, Pelupessy I, and Zwart SP. Multiscale and multi-domain computational astrophysics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2021), 2014.

34. Goddard WA, Merinov B, van Duin ACT, and Jacob T. Multi-paradigm multi-scale simulations for fuel cell catalysts and membranes. *Molecular Simulation*, 32(3–4):251–268, 2006.

35. Regenauer-Lieb K, Veveakis M, Poulet T, Wellmann F, Karrech A, Liu J, Hauser J, Schrank C, Gaede O, and Trefry M. Multiscale coupling and multiphysics approaches in Earth sciences: Theory. *Journal of Coupled Systems and Multiscale Dynamics*, 1(1):49–73, 2013.

36. Chopard B, Borgdorff J, and Hoekstra AG. A framework for multi-scale modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2021), 2014.

37. Uhrmacher AM, Ewald R, John M, Maus C, Jeschke M, and Biermann S. Combining micro and macro-modeling in DEVS for computational biology. In *Proceedings of the Winter Simulation Conference (WSC)*, 2007.

38. Santucci JF, Capocchi L, and Zeigler BP. System entity structure extension to integrate abstraction hierarchies and time granularity into DEVS modeling and simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 92(8):747–769, 2016.

39. golang.org. *The Go Programming Language: Packages (Software Version 1.6)*, 2016.

40. Varga A. *OMNeT++ User Manual (Version 4.5)*, 2014.

41. nsnam.org. *ns-3 Discrete-Event Network Simulator Tutorial (Software Version ns-3.25)*, 2016.

42. Goldberg D. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

43. Rapaport DC. *The Art of Molecular Dynamics*. Cambridge University Press, New York, NY, USA, second edition, 2004.

44. Shaw DE, Grossman JP, Bank JA, Batson B, Butts JA, Chao JC, Deneroff MM, Dror RO, Even A, Fenton CH, Forte A, Gagliardo J, Gill G, Greskamp B, Ho CR, Ierardi DJ, Iserovich L, Kuskin JS, Larson RH, Layman T, Lee LS, Lerer AK, Li C, Killebrew D, Mackenzie KM, Mok SYH, Moraes MA, Mueller R, Nociolo LJ, Peticolas JL, Quan T, Ramot D, Salmon JK, Scarpazza DP, Ben Schafer U, Siddique N, Snyder CW, Spengler J, Tang PTP, Theobald M, Toma H, Towles B, Vitale B, Wang SC, and Young C. Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

45. Hartmann P, Donkó Z, Lévai P, and Kalman GJ. Molecular dynamics simulation of strongly coupled QCD plasmas. *Nuclear Physics A*, 774:881–884, 2006.

46. Smolin L. Atoms of space and time. *Scientific American*, 23(4):94–103, 2014.

47. Vogelsberger M, Genel S, Springel V, Torrey P, Sijacki D, Xu D, Snyder GF, Bird S, Nelson D, and Hernquist L. Properties of galaxies reproduced by a hydrodynamic simulation. *Nature*, 509(7499):177–182, 2014.

48. Adams FC and Laughlin G. A dying universe: The long term fate and evolution of astrophysical objects. *Reviews of Modern Physics*, 69(2):337–372, 1997.

49. Wieland F. The threshold of event simultaneity. *Simulation: Transactions of the Society for Modeling and Simulation International*, 16(1):23–31, 1999.

50. Zeigler BP, Moon Y, and Kim D. High performance modelling and simulation: Progress and challenges. Technical report, University of Arizona, 1996.

51. Lee JJ. How do earthquake early warning systems work? Published online by National Geographic, 2013. Accessed on May 7, 2015 at http://news.nationalgeographic.com.

52. Josuttis NM. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, New York, NY, USA, second edition, 2012.

53. Nutaro JJ. *Building Software for Simulation: Theory and Algorithms with Applications in C++*. John Wiley & Sons, Hoboken, NJ, USA, 2011.

54. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

55. Jefferson DR. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7 (3):404–425, 1985.

56. Raynal M and Singhal M. Logical time: Capturing causality in distributed systems. *Computer*, 29(2): 49–56, 1996.

57. Rönngren R and Liljenstam M. On event ordering in parallel discrete event simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*, 1999.

58. Nutaro JJ and Sarjoughian HH. A unified view of time and causality and its application to distributed simulation. In *Proceedings of the Summer Computer Simulation Conference (SCSC)*, 2003.

59. Lee EA. Constructive models of discrete and continuous physical phenomena. *IEEE Access*, 2:797–821, 2014.

60. Barros F. Semantics of multisampling systems. *International Journal of Simulation and Process Modelling*, 11(5):374–389, 2016.

61. Goldstein R and Khan A. A taxonomy of event time representations. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS)*, 2017.

62. Rönngren R and Ayani R. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, 1997.

63. Vangheluwe HLM. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design (CACSD)*, 2000.

64. Karplus M. Development of multiscale models for complex chemical systems: From H+H2 to biomolecules (Nobel lecture). *Angewandte Chemie International Edition*, 53(38):9992–10005, 2014.

Appendix A Representation mathematics

Provided here are the mathematical operations and algorithms associated with the multiscale simulated time representation, which consists of the scale, duration, and time point data types, as well as the time sequence, time queue, and time cache data structures.

A.1 Scale mathematics

The scale data type represents the general concept of scale as a dimensionless power of 1000. Its primary role in the multiscale time representation is to specify a model's level of time precision as a base-1000 SI unit, satisfying Principles 1 and 2 of Section 4.1. We imagine that a scale value could also be used to characterize the magnitude of distance-valued quantities in a multiscale representation of space, though this is future work.

To reason mathematically about the scale data type, we express a scale value using the notation below. The symbol $\blacklozenge$ represents the factor of 1000 between scales, and the attribute *level* is an integer exponent.

$$\blacklozenge^{level} \qquad \{\text{scale } (1000^{level})\}$$

The custom notation offers several advantages over a conventional expression such as 1000^{level} or β^{level} (see Section 4.3 for the definition of β). The $\blacklozenge^{level}$ expression is more compact than 1000^{level}, and unlike β^{level} it clearly indicates the fact that scale has only a single attribute. More importantly, the custom notation allows us to restrict the set of mathematical operations that can be performed on scale values. For example, we consider the product $\blacklozenge^{level_1} \cdot \blacklozenge^{level_2}$ meaningless. However, it is useful to express the division of one scale by another to yield a power of 1000.

$$\frac{\blacklozenge^{level_1}}{\blacklozenge^{level_2}} = 1000^{level_1 - level_2}$$

An implementation of the scale data type is described in Appendix B.1.

A.2 Duration mathematics

The duration data type represents a quantity of time as a multiple of a time quantum, where the time quantum is based on a precision level represented by a value of type scale. The primary role of duration is to quantify the simulated time between events in a discrete-event simulation.

A duration value is expressed as follows, where m is the multiplier and $\blacklozenge^u$ is the precision level.

$$\text{◔}\, m\blacklozenge^u \qquad \{\text{duration } (m \cdot 1000^u \text{ s})\}$$

The quantity of time represented is $m \cdot 1000^u$ seconds. Examples of durations are given below.

◔$1\blacklozenge^0$	{1 second}
◔$10\blacklozenge^0$	{10 seconds}
◔$10\blacklozenge^{-1}$	{10 milliseconds}
◔$7158\blacklozenge^{-2}$	{7158 microseconds}
◔$-200\blacklozenge^{-2}$	{−200 microseconds}
$-$◔$200\blacklozenge^{-2}$	{−200 microseconds}
◔$200\blacklozenge^{1}$	{200 kiloseconds}

The standard arithmetic operators +, −, ·, and / are used to express fixed-point operations. The fixed-point addition (+) and subtraction (−) operations require both duration operands to have the same precision level, and the result is always a duration value with the same precision as the operands. If the resulting multiplier is less than 1000^5 in magnitude, addition and subtraction yield exact results.

◔$2\blacklozenge^0$ + ◔$3\blacklozenge^0$	{5 seconds}
◔$2\blacklozenge^0$ − ◔$3\blacklozenge^0$	{−1 second}

The fixed-point multiplication (·) and division (/) operations always involve one duration operand and one scalar. In the case of division, the duration must be the numerator. For both multiplication and division, the precision level of the resulting duration value is the same as the duration-valued operand. This essentially means that the multiplier m of the original duration is multiplied or divided by the scalar operand. In the examples below, no rounding is needed.

$$\begin{aligned} 5 \cdot \text{◔}\,100\blacklozenge^0 &= \text{◔}\,500\blacklozenge^0 \\ \tfrac{1}{5} \cdot \text{◔}\,100\blacklozenge^0 &= \text{◔}\,20\blacklozenge^0 \\ \text{◔}\,100\blacklozenge^0 \cdot \tfrac{1}{5} &= \text{◔}\,20\blacklozenge^0 \\ \text{◔}\,100\blacklozenge^0/5 &= \text{◔}\,20\blacklozenge^0 \end{aligned}$$

In the following case, the resulting multiplier is rounded from 12.5 to 13. A fraction of exactly half the time quantum is rounded away from zero, though other rounding conventions are possible.

$$\text{◔}\,100\blacklozenge^0/8 = \text{◔}\,13\blacklozenge^0$$

Comparison operations assume that the duration operands on both sides could be replaced by their associated quantities $m \cdot 1000^u$ s.

$$\begin{aligned} \text{◔}\,2\blacklozenge^0 &> \text{◔}\,1000\blacklozenge^{-1} \\ \text{◔}\,2\blacklozenge^0 &< \text{◔}\,3000\blacklozenge^{-1} \\ \text{◔}\,-8\blacklozenge^{-4} &< \text{◔}\,-7\blacklozenge^{-4} \end{aligned}$$

An important consequence is that two or more duration values may be equal without being the same. For example, the three values below are equal since they all represent 1 s. Yet the values are distinct since they have different precision levels.

$$\text{◔}\,1\blacklozenge^0 = \text{◔}\,1000\blacklozenge^{-1} = \text{◔}\,1000000\blacklozenge^{-2}$$

The possibility of duration values being equal yet distinct has significant implications. For example, applying the same fixed-point operation to the three versions of 1 s yields different results.

$$\begin{aligned} \tfrac{1}{3} \cdot \text{◔}\,1\blacklozenge^0 &= \text{◔}\,0\blacklozenge^0 \\ \tfrac{1}{3} \cdot \text{◔}\,1000\blacklozenge^{-1} &= \text{◔}\,333\blacklozenge^{-1} \\ \tfrac{1}{3} \cdot \text{◔}\,1000000\blacklozenge^{-2} &= \text{◔}\,333333\blacklozenge^{-2} \end{aligned}$$

Whereas the = comparison operator tests only equality, one may use the ≡ operator to indicate that two duration values are equivalent in both multiplier and time precision.

$$\begin{aligned} \text{◔}\,500\blacklozenge^{-1} + \text{◔}\,500\blacklozenge^{-1} &= \text{◔}\,1\blacklozenge^0 \\ \text{◔}\,500\blacklozenge^{-1} + \text{◔}\,500\blacklozenge^{-1} &\equiv \text{◔}\,1000\blacklozenge^{-1} \end{aligned}$$

If the multiplier of a computed duration reaches or exceeds the limit of 1000^5 as a result of a fixed-point operation, it overflows yielding a positive or negative infinite duration.

$$\begin{aligned} \text{◔}\,999999999999999\blacklozenge^2 + \text{◔}\,1\blacklozenge^2 &= \text{◔}\,\infty \\ -1000000 \cdot \text{◔}\,1000000000\blacklozenge^{-5} &= \text{◔}\,-\infty \end{aligned}$$

While we regard the duration data type as a fixed-point representation of time, an implementation effort revealed the benefits of accommodating floating-point operations when needed. Taking inspiration from Goldberg [42], we use $\oplus$, $\ominus$, $\odot$, and $\oslash$ to indicate floating-point arithmetic. However, in the case of duration, these are not base-2 but rather base-1000 operations. Base-1000 floating-point operations support arithmetic between duration values with different precision levels, as in the following examples.

$$\begin{aligned} \text{◔}\,3\blacklozenge^0 \oplus \text{◔}\,475\blacklozenge^{-1} &\equiv \text{◔}\,3475\blacklozenge^{-1} \\ \text{◔}\,1\blacklozenge^1 \oplus \text{◔}\,1\blacklozenge^{-2} &\equiv \text{◔}\,1000000001\blacklozenge^{-2} \\ \text{◔}\,500\blacklozenge^{-4} \ominus \text{◔}\,1\blacklozenge^{-3} &\equiv \text{◔}\,-500\blacklozenge^{-4} \end{aligned}$$

The above examples yield exact results, though floating-point operations can be used for approximation as seen below.

$$\begin{aligned} \tfrac{1}{3} \odot \text{duration}(1, u{=}0) &= \text{duration}(333333333333333, u{=}-5) \\ \tfrac{1}{3} \odot \text{duration}(1000, u{=}-1) &= \text{duration}(333333333333333, u{=}-5) \\ \text{duration}(1000, u{=}-1) \oslash 3 &= \text{duration}(333333333333333, u{=}-5) \end{aligned}$$

Floating-point operations on duration values select the time precision of the result that achieves the highest accuracy possible given the 1000^5 multiplier limit and the base of 1000. If there are multiple resulting precision levels that would yield the highest achievable accuracy, then the chosen precision is the one which differs least from the finest precision among the operands. These rules allow precision levels to be adjusted to avoid overflow, as shown below.

$$\begin{aligned} &\text{duration}(999999999999999, u{=}2) \oplus \text{duration}(1, u{=}2) \\ &\qquad \equiv \text{duration}(1000000000000, u{=}3) \end{aligned}$$

$$\begin{aligned} &-1000000 \odot \text{duration}(1000000000, u{=}-5) \\ &\qquad \equiv \text{duration}(-1000000000000, u{=}-4) \end{aligned}$$

The division of one duration by another is always considered a floating-point operation, even if the $\oslash$ operator does not appear. As expressed by the general rule below, the numerator and denominator may have different precision levels and the result is a scalar with no prescribed precision. Implementations should represent the result using a standard double precision floating-point value.

$$\frac{\text{duration}(m_1, u{=}u_1)}{\text{duration}(m_2, u{=}u_2)} = \frac{m_1}{m_2} \cdot 1000^{u_1 - u_2} \qquad \{\text{general rule}\}$$

$$\frac{\text{duration}(1, u{=}-1)}{\text{duration}(1, u{=}0)} = 0.001 \qquad \{\text{example}\}$$

Although the floating-point operations may seem convenient, simulation frameworks should be designed to discourage their use. Our multiscale approach relies on consistent precision levels within atomic models, and requires exact addition and subtraction in the simulator. The fixed-point operations have these properties.

There is one last detail concerning the mathematics of the duration data type. We have seen that a positive or negative infinite duration has no time precision, as it is not needed. Intuitively, a duration of zero should not require a precision level either. Yet we do in fact include the time precision for zero durations. Although 0 s, 0 ms, and 0 μs all represent the same quantity of simulated time, the associated precision levels are significant in that they may affect subsequent fixed-point operations.

$\text{duration}(\infty)$	{infinite duration (no time precision)}
$\text{duration}(-\infty)$	{negative infinite duration}
$\text{duration}(0, u{=}0)$	{zero seconds}
$\text{duration}(0, u{=}-1)$	{zero milliseconds}
$\text{duration}(0, u{=}-2)$	{zero microseconds}

Our implementation of the duration data type makes use of a C++11 feature allowing SI units to appear as part of the syntax. For example, the code `5_ms` represents 5 milliseconds. Further implementation-related details are provided in Appendix B.2.

A.3 Time point mathematics

The time point data type represents points in simulated time. It is intended for use in simulator code where, as emphasized by Principle 4 of Section 4.1, software complexity is tolerated to a greater extent than in model code. The main purpose of the time point is to describe event times as offsets from a common reference point. Because the reference point is arbitrary, it makes little sense to perform addition or subtraction on pairs of time point instances, not to mention multiplication or division. Rather, the key operations involve either perturbing a time point instance using a duration value, or obtaining a duration value expressing the difference between two time point instances.

A time point instance contains an arbitrary-precision integer in the form of a vector of n base-1000 digits $d_0 \ldots d_{n-1}$, plus a scale value representing the time precision associated with the least significant digit. The time point notation is shown below.

$$\text{time\_point}([\underset{n-1}{d}, \ldots, \underset{2}{d}, \underset{1}{d}, \underset{0}{d}], u) \qquad \{\text{time point}\}$$

Underneath each digit is a corresponding index. The little-endian convention is used, so the least significant digit d_0 has the smallest index. Each digit is an integer in the range $0 \le d_i < 1000$, and the represented time offset in seconds is given by the following expression.

$$\sum_{i \in \{0,\ldots,n-1\}} d_i \cdot 1000^{u+i} \qquad \{\text{offset in seconds}\}$$

Below are several examples of time point instances. The meaning of each expression is shown in hours (HH), minutes (MM), seconds (SS), and fractions of sections (xx...): HH:MM:SS.xxxxxx. The empty vector, shown as $\text{time\_point}([], u{=}0)$, represents 00:00:00.

$\text{time\_point}([], u{=}0)$	{00:00:00}
$\text{time\_point}([\underset{0}{45}], u{=}0)$	{00:00:45}
$\text{time\_point}([\underset{1}{45}, \underset{0}{1}], u{=}-1)$	{00:00:45.001}
$\text{time\_point}([\underset{2}{45}, \underset{1}{1}, \underset{0}{92}], u{=}-2)$	{00:00:45.001092}
$\text{time\_point}([\underset{2}{645}, \underset{1}{1}, \underset{0}{92}], u{=}-2)$	{00:10:45.001092}
$\text{time\_point}([\underset{3}{504}, \underset{2}{645}, \underset{1}{1}, \underset{0}{92}], u{=}-2)$	{140:10:45.001092}

Unlike a floating-point value, a time point instance can represent an extremely large magnitude yet still resolve arbitrarily small differences. For example, the sum of a

yettasecond (10^{24} s) and a yoctosecond (10^{-24} s) can be exactly represented.

$$[1, 0, 0, \ldots, 0, 0, 1]^{-8} \quad \{10^{24}\,\mathrm{s} + 10^{-24}\,\mathrm{s}\}$$

A finite duration value can be added to or subtracted from a time point instance. The result is always exact, regardless of the magnitude or precision of either operand. Note that the vector of digits in the resulting time point may expand to accommodate any combination of scales.

$$[\,]^{0} + 5000388^{-3} = [5, 0, 388]^{-3}$$

$$[5, 0, 388]^{-3} + 1777^{-4} = [5, 0, 389, 777]^{-4}$$

The vector of digits also shrinks if possible. Any zero digits on either end are removed. This may result in a coarsening of the time precision, as in the second example below.

$$[5, 600, 280, 777]^{-4} - 5^{-1} = [600, 280, 777]^{-4}$$

$$[600, 280, 777]^{-4} + 223^{-4} = [600, 281]^{-3}$$

When the time point data type is used in a simulation to express the current point in simulated time, we do not use straight-forward addition operations to advance the time point instance. Rather, we apply multiscale time advancement. Given a time point instance t and a finite, nonnegative duration value m^{u}, multiscale time advancement is denoted $t \triangleright m^{u}$. If the advancement duration is zero ($m = 0$), the result is same as the original time point. If the duration is positive, the result is a time point instance that is similar to $t + m^{u}$ except truncated at the precision level of the duration. Essentially, all digits less significant than ^{u} are discarded.

Advancing equal time point instances by equal durations may yield different outcomes if the durations have different precision levels. This effect is shown in the examples below. Note that the duration is equal to 1150 ms in all three cases.

$$[72, 800, 444, 321]^{-3} \triangleright 1150000000^{-3} = [73, 950, 444, 321]^{-3}$$

$$[72, 800, 444, 321]^{-3} \triangleright 1150000^{-2} = [73, 950, 444]^{-2}$$

$$[72, 800, 444, 321]^{-3} \triangleright 1150^{-1} = [73, 950]^{-1}$$

There are two ways to measure the difference between two time point instances. The first is regular subtraction, using the $-$ operator. The second is an approximation of subtraction we refer to as a *gap* operation and express using $\ominus$. The rational for providing these two similar operations is that they both a yield duration value, and the duration data type cannot always represent the exact result. In some cases we are willing to approximate the difference between two time point instances, so we use one operation; sometimes we are not willing to employ approximation, so we use the other operation.

If the exact difference between two time points can be expressed with a multiplier less than 1000^5, then $-$ and $\ominus$ produce the same result.

$$[31, 775\,100]^{-2} - [1, 170]^{-1} = 30605100^{-2}$$

$$[31, 775\,100]^{-2} \ominus [1, 170]^{-1} = 30605100^{-2}$$

Below are two more examples of $-$ and $\ominus$ producing the same result. Although the initial time point spans more than 1000^5 quanta, the subtraction eliminates a digit.

$$[7, 3, 5, 6, 2, 9]^{-5} - [7]^{0} = 3005006002009^{-5}$$

$$[7, 3, 5, 6, 2, 9]^{-5} \ominus [7]^{0} = 3005006002009^{-5}$$

If the exact difference between the time points cannot be represented with a multiplier less than 1000^5, subtraction yields an infinite duration whereas the gap operation produces an approximation. In the $\ominus$ operation below, for example, the result is off by 9 fs.

$$[7, 3, 5, 6, 2, 9]^{-5} - [6]^{0} = \infty$$

$$[7, 3, 5, 6, 2, 9]^{-5} \ominus [6]^{0} = 1003005006002^{-4}$$

We require the result of a gap operation to be within a single time quantum of the resulting duration value. Consider the formulas below, which suggest different results for the same $\ominus$ operation. The first result ($=$) has an error of 0.2 fs. It is the best possible approximation, and the error is less than the 1 fs time quantum of the resulting duration. The second result ($\approx$) has a much larger error of 2.8 fs. It is still acceptable, however, since the time quantum of the result is now 1 ps. The third result ($\neq$) has an error of 1.8 fs. Although this is a closer

approximation than the second result, it is invalid since the error is not less than the resulting duration's time quantum of 1 fs.

$$\begin{aligned} \text{clock}[\underset{0}{1}]\diamond^{0} \ominus \text{clock}[\underset{0}{2800}]\diamond^{-6} &= \text{pie}\,999999999999997\diamond^{-5} \\ \text{clock}[\underset{0}{1}]\diamond^{0} \ominus \text{clock}[\underset{0}{2800}]\diamond^{-6} &\approx \text{pie}\,100000000000\diamond^{-4} \\ \text{clock}[\underset{0}{1}]\diamond^{0} \ominus \text{clock}[\underset{0}{2800}]\diamond^{-6} &\neq \text{pie}\,999999999999999\diamond^{-5} \end{aligned}$$

$$\{\text{exact}: \quad 1\,\text{s} - 2800\,\text{as} = 999999999999997200\,\text{as}\}$$

There is actually a second rule concerning $\ominus$—a somewhat obvious rule—which is that $t_\mathsf{A} \ominus t_\mathsf{B}$ must never be approximated as zero if time points t_A and t_B differ.

The above rules concerning gap operations ensure that multiscale time advancement has a convenient property. Suppose we have time point instances t_A and t_B, where $t_\mathsf{A} < t_\mathsf{B}$. Also suppose that we introduce a mutable variable t, and assign it an initial value of t_A. If we repeatedly advance t toward t_B using the assignment $t \leftarrow t \triangleright (t_\mathsf{B} \ominus t)$, then t will eventually reach t_B. In practice, only one or two iterations will be required, but the important point is that there exists a mechanism to advance any time point to a future time point.

Taking the rules a step further, the gap operation should be implemented such that the resulting duration value's precision is always the one that yields the highest achievable degree of accuracy while satisfying the above requirements. If there are still multiple levels to choose from, one should select the level closest to the finest precision among the operands. If followed, these additional conventions will help standardize multiscale time value operations.

Appendix B.3 outlines how the data structure can be implemented in C++.

A.4 Time sequence mathematics

The time sequence data structure stores a set of unique time points that are appended in increasing order. The structure's primary role is to aid in the recording of simulation results. For example, the recorded time points may represent event times or the endpoints of time series segments. A simple vector of time point instances would fulfill the same role, but what distinguishes a time sequence is that where possible, time points are stored using duration values to save memory. The fact that exact time points are recorded satisfies Principle 6 of Section 4.1, while the use of 64-bit duration values addresses the efficiency considerations of Principle 7.

Suppose one wishes to store time points t_A, t_B, and t_C, defined below.

$$\begin{aligned} t_\mathsf{A} &= \text{clock}[\underset{0}{5}]\diamond^{-2} \\ t_\mathsf{B} &= \text{clock}[\underset{3}{5}, \underset{2}{0}, \underset{1}{0}, \underset{0}{72}]\diamond^{-5} \\ t_\mathsf{C} &= \text{clock}[\underset{6}{3}, \underset{5}{600}, \underset{4}{0}, \underset{3}{5}, \underset{2}{0}, \underset{1}{0}, \underset{0}{72}]\diamond^{-5} \end{aligned}$$

The resulting time sequence instance would store t_A as a full time point instance, then attempt to record t_B and t_C as offsets from t_A. This is possible in the case of t_B, since $t_\mathsf{B} - t_\mathsf{A}$ can be represented as a duration value. However $t_\mathsf{C} - t_\mathsf{A}$ cannot be represented as a duration since the multiplier would not be less than 1000^5:

$$\begin{aligned} t_\mathsf{B} - t_\mathsf{A} &= \text{pie}\,72\diamond^{-5} \\ t_\mathsf{C} - t_\mathsf{A} &= \text{pie}\,\infty \end{aligned}$$

In order to record t_C, it is necessary to store a second time point instance. Ultimately, the following information would be stored: a sequence of offsets (of type duration), and a typically shorter sequence of partitions. Each partition would contain two elements: the index of the first of a group of offsets which belong to the partition, and the time point instance to which the offsets must be added. For example, the sequence t_A, t_B, t_C would be represented using the three offsets shown below, which are grouped into partitions $[0, t_\mathsf{A}]$ and $[2, t_\mathsf{C}]$.

$$\underbrace{\text{pie}\,\underset{0}{0}\diamond^{0}, \text{pie}\,\underset{1}{72}\diamond^{-5}}_{[0,\,t_\mathsf{A}]}, \underbrace{\text{pie}\,\underset{2}{0}\diamond^{0}}_{[2,\,t_\mathsf{C}]}$$

The following information would be stored:

$$\left[\text{pie}\,0\diamond^{0}, \text{pie}\,72\diamond^{-5}, \text{pie}\,0\diamond^{0}\right] \qquad \{\textit{offsets}\}$$

$$\left[0, \text{clock}[\underset{0}{5}]\diamond^{-2}\right] \qquad \{\textit{partition}[0]\}$$

$$\left[2, \text{clock}[\underset{6}{3}, \underset{5}{600}, \underset{4}{0}, \underset{3}{5}, \underset{2}{0}, \underset{1}{0}, \underset{0}{72}]\diamond^{-5}\right] \qquad \{\textit{partition}[1]\}$$

Algorithm 1 formalizes the procedure for appending a time point t onto a time sequence. The key decision is on line 11, where the algorithm determines if t can be appended as a duration offset Δt (line 12) or if a new partition is needed (lines 14 and 15).

The opposing procedure, obtaining the time point at a given index i, is relatively straightforward. One may use a binary search to identify the encompassing partition $[i_p, t_p]$, after which $t_p + \textit{offsets}[i]$ yields the time point in question.

For most applications, a time sequence instance will have only a single partition. The performance penalty of storing and searching through arbitrary-precision time point instances should surface only in multiscale contexts where the added complexity is justified.

Algorithm 1 Time Sequence Append

```
1: function APPEND(time_sequence, t)
2:     [partitions, offsets] ← time_sequence
3:     n_p ← #partitions
4:     n ← #offsets
5:     Δt ← ◔∞
6:     if n_p > 0 then
7:         [i, t_i] ← partitions[n_p − 1]
8:         Δt ← t − t_i
9:         ASSERT(Δt > offsets[n − 1])
10:    end if
11:    if Δt < ∞ then
12:        offsets ← offsets ‖ [Δt]
13:    else
14:        partitions ← partitions ‖ [n, t]
15:        offsets ← offsets ‖ [◔0◆^0]
16:    end if
17:    return [partitions, offsets]
18: end function
```

The time sequence data structure could be modified to allow the removal of recently added time points, which would support roll-back operations in optimistic parallel simulation. Insertions and removals in the middle of a sequence are not possible unless the underlying algorithm is replaced. Fortunately, these operations are rarely encountered in discrete-event simulation, since any alteration in behavior at time t is likely to invalidate results obtained later than t.

Appendix B.4 shows how custom iterators can be incorporated into a C++ implementation of the time sequence data structure, and how the structure supports a relational approach to storing simulation results.

A.5 Time queue mathematics

The time queue data structure stores the current point in simulated time, and keeps track of future time points when events are scheduled to occur. It is the most sophisticated element of the proposed time representation. The time queue ensures each atomic model perceives time according to its own precision level (Principle 5, Section 4.1). Yet being an element of the simulator, it maintains exact timing information (Principle 6).

The time queue tracks future event times not as time point instances, but rather as duration values (Principle 7). It is similar to the time sequence in this regard, providing arbitrary precision despite using a memory-efficient representation for internally stored event times. However, the time queue solves a considerably more difficult problem. First, future event times may be added to the queue in any order, provided they do not precede the current time. Second, future event times may also be canceled before they have a chance to occur. Third, numerous comparisons between event times are needed as part of the searching and sorting inherent in any priority queue; these comparison operations must be performed efficiently without allocating memory in the form of extra time point digits. As explained in Section 4.2.4, our solution involves tracking every event time as a planned phase: an offset from the reference point at the beginning of an encompassing epoch. We use the term "phase" as a shorthand for "planned phase". This concept, which pertains to scheduled events, must not to be confused with the "scale phase" or "epoch phase" quantities defined below, which pertain to the current time.

The key to the approach is the conversion between planned durations and phases. To schedule an event, a planned duration must be converted into a phase to be stored in the queue. To advance time to the most imminent scheduled event, the stored phase must be converted into a planned duration. Let us formalize these conversions given a current time t and a scale of interest $◆^u$. We start by selecting notations for two quantities:

$$\langle t \rangle_{◆^u} \qquad \{\text{scale phase}\}$$

$$\langle t \rangle_{◆^u}^{◆^{u+\eta}} \qquad \{\text{epoch phase}\}$$

The *scale phase* is essentially the base-1000 digit of t at the scale of interest, or zero if no digit exists at that scale. This definition assumes t is never negative, as otherwise some arithmetic is needed such that the offset represented by the digit is measured away from $-\infty$ as opposed to zero. As defined in Section 4.2.4, the epoch phase is the number of time quanta separating t from the beginning of the current epoch. With $\beta = 1000$ and $\eta = 5$ (see Section 4.3), the two quantities are bounded as follows:

$$0 \;\le\; \langle t \rangle_{◆^u} \;<\; \beta$$

$$0 \;\le\; \langle t \rangle_{◆^u}^{◆^{u+\eta}} \;<\; \beta^\eta$$

The epoch phase can be derived from scale phases:

$$\langle t \rangle_{◆^u}^{◆^{u+\eta}} \;=\; \sum_{i \in \{0,\ldots,\eta-1\}} \beta^i \cdot \langle t \rangle_{◆^{u+i}}$$

The procedure for converting a planned duration Δt into a planned phase Δt_ϕ is given in Algorithm 2. The decision point on line 4 checks whether the event is in the next epoch instead of the current one.

Algorithm 3 converts a planned phase Δt_ϕ into a planned duration Δt. Line 4 again checks whether the event is in the next epoch.

Note that the multipliers m and m_ϕ and the epoch phase in Algorithms 2 and 3 are regular integers, not special data types. Hence the addition and subtraction operations involving these variables are just standard arithmetic operations which require no rounding.

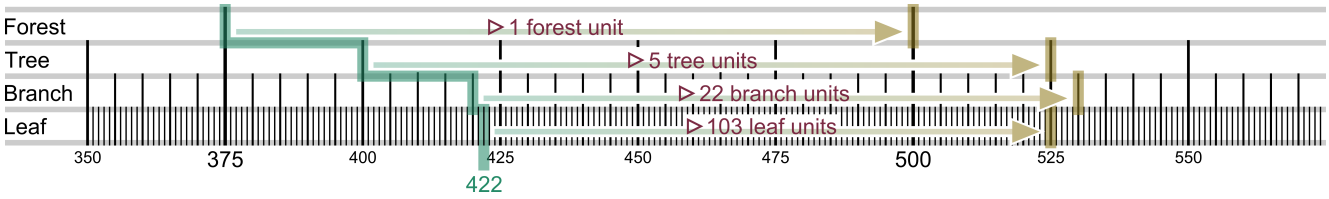


Figure 19. The time queue shown features one scheduled event at each of the four scales. As a result of multiscale time advancement, scheduled event times do not necessarily adhere to the order implied by their respective planned durations. Note that in general, a time queue can have multiple scheduled events at each scale.

Algorithm 2 Basic Conversion to Phase

```
1: function PHASE_FROM_DURATION(t, Δt)
2:     (m)^u ← Δt
3:     m_φ ← ⟨t⟩_{◆^u}^{◆^{u+5}} + m
4:     if m_φ ≥ 1000^5 then
5:         m_φ ← m_φ − 1000^5
6:     end if
7:     Δt_φ ← (m_φ)^u
8:     return Δt_φ
9: end function
```

Algorithm 3 Basic Conversion to Duration

```
1: function DURATION_FROM_PHASE(t, Δt_φ)
2:     (m_φ)^u ← Δt_φ
3:     m ← m_φ − ⟨t⟩_{◆^u}^{◆^{u+5}}
4:     if m < 0 then
5:         m ← m + 1000^5
6:     end if
7:     Δt ← (m)^u
8:     return Δt
9: end function
```

Until now we have assumed a common time precision for all planned durations used to schedule events. Yet the objective is to support atomic models with diverse time scales. The precision levels of the planned durations may therefore differ, and so the time queue must acknowledge the existence of not just two epochs (one "current" and one "next"), but several distinct pairs of epochs (one "current" and one "next" at each scale).

The incorporation of multiple precision levels into a single queue is further complicated by the fact that a planned duration Δt_p operates on the current time t using multiscale time advancement. The single-scale scenario described in Figure 14 of Section 4.2.4 suggests that $t + \Delta t_p$ gives the event time, but this is only true if there is one time precision. The general expression for each event time is in fact $t \triangleright \Delta t_p$. As a result, the actual order of scheduled event times may be inconsistent with their planned durations.

Consider the scenario in Figure 19. As in the Section 4.2.4 scenario, the current time is 422 leaf units, but here it is shown as a jagged line to reflect the inclusion of durations expressed in branch, tree, and forest units. Observe that the forest and tree scales feature scheduled events with equal planned durations (1 forest unit = 5 tree units). Surprisingly, the forest-scale event is scheduled at an earlier time. The branch scale event has a shorter Δt_p (22 branch units < 5 tree units), but is scheduled at a later time. Finally, the tree- and leaf-scale events are both scheduled for time 525 despite having different planned durations.

Although scenarios involving multiple precision levels are conceptually intimidating, only a few enhancements are needed to make the time queue function properly. First, in order to identify and properly handle simultaneous events, we need to ensure that every unique future event time has a unique planned phase. We accomplish this by coarsening the phase; the time precision is repeatedly increased by one scale so long as the event time does not suffer any rounding error. In Figure 19, the 103 leaf unit event has a time of 525, which can be coarsened to the tree scale without losing precision. This leaf-scale event is then stored with the 5 tree unit event as a group of two simultaneous events. With such groups identified, any of a number of simultaneous event handling techniques can be adopted. A tie-breaking function can be used, as in the classic version of the DEVS formalism, or the grouped events can be executed concurrently, as in a variant of that formalism called Parallel DEVS[3].

For this first enhancement, Algorithm 2 is replaced by Algorithm 4 when converting planned durations into planned phases. The key difference is the loop on line 10 of Algorithm 4, which repeatedly coarsens the precision by one scale so long as no accuracy is lost. Once accuracy would be lost, the *maximized* flag is set and the loop is terminated. This ensures the coarsest possible prevision level is used to store each future event time.

Algorithm 4 is complicated by two special cases. If a scheduled event occurs at $t = 0$, then the phase precision can be coarsened indefinitely. The *unbounded* flag detects this case, and on line 30 the default precision level $\blacklozenge^{u_{default}}$ is adopted for the planned phase. It is safe to use $u_{default} = 0$ if the simulation starts at time zero. The other special case is a planned duration of zero, which is problematic if its precision level is too coarse. We detect this case on line 4, and on line 5 we re-initialize

Algorithm 4 Enhanced Conversion to Phase

1: **function** PHASE_FROM_DURATION(t, Δt)
2: $\langle t \rangle_{n-1}^{d_{n-1}, \ldots, d_0} \leftarrow t$
3: $m \leftarrow \Delta t$
4: **if** $m = 0$ **then**
5: $u \leftarrow u_t$
6: **end if**
7: $m_\phi \leftarrow \langle t \rangle_{u}^{u+5} + m$
8: *maximized* $\leftarrow \bot$
9: *unbounded* $\leftarrow \bot$
10: **while** $\neg$*maximized* $\wedge$ $\neg$*unbounded* **do**
11: *carry* $\leftarrow 0$
12: **if** $m_\phi \geq 1000^5$ **then**
13: $m_\phi \leftarrow m_\phi - 1000^5$
14: *carry* $\leftarrow 1$
15: **end if**
16: **if** MOD$(m_\phi, 1000) \neq 0$ **then**
17: *maximized* $\leftarrow \top$
18: **else if** $(m_\phi = 0) \wedge (u + 5 \geq u_t + n)$ **then**
19: **if** *carry* $= 0$ **then**
20: *unbounded* $\leftarrow \top$
21: **end if**
22: **end if**
23: **if** $\neg$*maximized* $\wedge$ $\neg$*unbounded* **then**
24: $m_\phi \leftarrow m_\phi / 1000$
25: $m_\phi \leftarrow m_\phi + 1000^4 \cdot \left(\langle t \rangle_{u+5} + \mathit{carry} \right)$
26: $u \leftarrow u + 1$
27: **end if**
28: **end while**
29: **if** *unbounded* **then**
30: $u \leftarrow u_{default}$
31: **end if**
32: $\Delta t_\phi \leftarrow m_\phi$
33: **return** Δt_ϕ
34: **end function**

the time precision to that of the current time. It will then be coarsened, but the *maximized* flag will prevent it from becoming coarser than the phases of other events scheduled for the same time.

The second time queue enhancement is a means of comparing planned phases with different precision levels. The naïve approach is to simply convert every phase into a time point instance, but this would likely sacrifice the efficiency gains that motivated the use of phases in the first place. Instead, we need a fixed-memory algorithm for comparing two planned phases based on the future event times they represent. Given two planned phases with different precision levels, the method for sorting their respective future event times is as follows.

1. Convert both planned phases into planned durations using Algorithm 3.
2. If one planned duration has a coarser time precision than the other, refine it until the precision levels match.
3. After refinement, the shorter planned duration corresponds to the more imminent future event.

The operation in Step 2 can be achieved using Algorithm 5, which refines one planned duration so that its relatively coarse time precision u matches another planned duration's precision $u_{refined}$. As usual, the current time t plays a critical role.

Algorithm 5 Planned Duration Refinement

1: **function** REFINED_DURATION(t, Δt, $u_{refined}$)
2: $m \leftarrow \Delta t$
3: **if** $m > 0$ **then**
4: **while** $(m < 1000^5) \wedge (u \geq u_{refined})$ **do**
5: $m \leftarrow 1000 \cdot m - \langle t \rangle_{u}$
6: $u \leftarrow u - 1$
7: **end while**
8: **end if**
9: $\Delta t' \leftarrow \infty$
10: **if** $m < 1000^5$ **then**
11: $\Delta t' \leftarrow m$
12: **end if**
13: **return** $\Delta t'$
14: **end function**

The final enhancement to support multiple scales allows one to take an event's planned duration Δt, and express it in the precision level $u_{rescaled}$ at which the event was originally scheduled. The first step is to use Algorithm 3 to convert the stored phase into the planned duration Δt. Algorithm 6 then completes the operation. Under normal circumstances, the phase will be at least as coarse as $u_{rescaled}$, and in this case line 5 re-purposes a function we have already defined. But for an event time of zero, the planned phase may not be coarse enough due to line 30 of Algorithm 4. In that case, the initial value on line 3 of Algorithm 6 will ultimately be returned.

Algorithm 6 Planned Duration Rescaling

1: **function** RESCALED_DURATION(t, Δt, $u_{rescaled}$)
2: $m \leftarrow \Delta t$
3: $\Delta t' \leftarrow m$
4: **if** $u \leq u_{rescaled}$ **then**
5: $\Delta t' \leftarrow$ REFINED_DURATION$(t, \Delta t, u_{rescaled})$
6: **end if**
7: **return** $\Delta t'$
8: **end function**

The methods and algorithms above (a) allow future events to be scheduled using planned durations featuring different precision levels, (b) allow the event times to be stored using phase durations that need not change as time advances, (c) allow the event times to be compared without constructing the corresponding time points, and (d) allow an event's planned duration to be expressed at its original precision level. This is all that is needed

to incorporate vastly different time scales into a single queue of future events.

To implement the time queue data structure, the C++ functions described in Appendix B.5 proved useful.

A.6 Time cache mathematics

One of the key insights underlying the DEVS formalism is the idea that the post-event state of a model instance may depend not only on the previous state, but possibly also on the duration of time elapsed since the previous event[3]. Thus if a simulation framework is to be as general as possible, a mechanism is needed to store and retrieve elapsed durations. The time cache data structure fulfills this role, providing elapsed durations by tracking previous events. It can be regarded as the opposite of the time queue, which tracks future events and provides planned durations.

Recall from Section 4.2.4 that for efficiency reasons, we dismissed the idea of storing many arbitrary-precision time point instances in the time queue. We want to minimize arbitrary-precision arithmetic in the time cache as well. Also recall that the time queue does not store planned durations directly, for then they would all need to be decreased whenever the current time advances. Similarly, the time cache should not store elapsed durations directly, as they would all need to be increased at every time advancement. Recall that the time queue works by storing planned phases measured from the beginning of a relevant epoch. A similar mechanism is needed for the time cache.

Fortunately, as explained in Section 4.2.5, the time cache data structure can be conveniently implemented by encapsulating and re-purposing the time queue. The idea is to track an imaginary future event instead of the actual past event. The actual event and its imaginary counterpart are always separated by the maximum representable duration, $(1000^5 - 1)\diamondsuit^u$, so it is a trivial matter to derive the time point of one from the time point of the other.

The mathematics underlying the time cache is as follows. Let Δt be the elapsed duration, measured between the previous event and the current time. Let $\Delta\tilde{t}$ be the *imaginary planned duration*, measured between the current time and the imaginary future event in the encapsulated time queue. We focus on a model instance with a time precision of $\diamondsuit^u$. When the instance undergoes an event, that event becomes a past event with a current elapsed duration of zero, or $0\diamondsuit^u$. The imaginary planned duration $\Delta\tilde{t}$ is then initialized as follows with the maximum multiplier of $1000^5 - 1$.

$$\Delta t = 0\diamondsuit^u \quad \Rightarrow \quad \Delta\tilde{t} = (1000^5 - 1)\diamondsuit^u$$

As time passes, the imaginary planned duration decreases. Suppose that the multiplier is $\tilde{m}$ at some point in time. If the elapsed duration Δt is then needed, it is calculated according to the formula below.

$$\Delta\tilde{t} = \tilde{m}\diamondsuit^u \quad \Rightarrow \quad \Delta t = (1000^5 - 1 - \tilde{m})\diamondsuit^u$$

An imaginary future event is terminated in any of three circumstances. First, the simulation may end, in which case the imaginary event is no longer needed. Second, the instance may undergo another event, which produces a new imaginary event, which replaces the existing one. Third, the current time may surpass the imaginary future event, meaning that the elapsed duration has increased beyond the maximum representable duration of $1000^5 - 1$ time quanta. In this last case, the imaginary future event is removed from the encapsulated time queue, and the corresponding past event is no longer tracked. If the elapsed duration is then needed for that model instance, it is reported as ∞. This is the mechanism by which infinite elapsed duration arise. As discussed in Section 5.1.3, an elapsed duration of ∞ means the model instance is in a steady state; otherwise, the modeler should not have allowed the instance to remain passive for 10^{15} or more time quanta.

A C++ implementation of the time cache data structure is briefly outlined in Appendix B.6.

Appendix B Representation implementation

The multiscale time representation consists of the six elements described in Section 4 and Appendix A: scale, duration, time point, time sequence, time queue, and time cache. The implementation of each element is outlined here with the aid of simplified C++11 code listings from our prototype simulator.

B.1 Scale implementation

In C++, the scale data type of Appendix A.1 is easily implemented as a `scale` class encapsulating the *level* attribute as the integer `level_`. The sample code below is a simplified version of the class declaration in our implementation. Comparison operators, string conversion functions, and selected `constexpr` qualifiers have been omitted.

```
1  class scale
2  {
3  public:
4    constexpr scale(int64 level);
5
6    int64 level() const;
7
8    scale& operator++();
9    scale& operator--();
10   scale& operator+=(int64);
11   scale& operator-=(int64);
12
13   scale operator+() const;
```

```
14    scale operator-() const;
15
16    scale operator+(int64) const;
17    scale operator-(int64) const;
18    int64 operator-(scale) const;
19    float64 operator/(scale) const;
20
21  private:
22    int8 level_;
23  };
```

Instead of constructing a `scale` value by invoking the constructor directly, the programmer may simply use one of the pre-defined values below.

```
constexpr scale yocto = scale(-8);
constexpr scale zepto = scale(-7);
constexpr scale atto  = scale(-6);
constexpr scale femto = scale(-5);
constexpr scale pico  = scale(-4);
constexpr scale nano  = scale(-3);
constexpr scale micro = scale(-2);
constexpr scale milli = scale(-1);
constexpr scale unit  = scale(0);
constexpr scale kilo  = scale(1);
constexpr scale mega  = scale(2);
constexpr scale giga  = scale(3);
constexpr scale tera  = scale(4);
constexpr scale peta  = scale(5);
constexpr scale exa   = scale(6);
constexpr scale zetta = scale(7);
constexpr scale yotta = scale(8);
```

Scales smaller than `yocto` (1000^{-8}) or larger than `yetta` (1000^{8}) must be obtained via the constructor (e.g. `scale(-9)`). The smallest and largest possible scales are determined by the integer data type encapsulated by `scale`. Observe that on line 22 of the class declaration, `level_` is stored as a signed 8-bit integer which accommodates all integers from -128 through 127. At the time of writing, it is difficult to imagine a need for scales smaller than 1000^{-128} or larger than 1000^{127}, regardless of whether time or space is the dimension of interest. Nevertheless, a larger integer data type could be chosen should the need arise.

Some operations are redundant in a mathematical context, yet convenient in a programming context. In mathematical notation, we disallow $\blacklozenge^{level_1} + 1$ as one can always write $\blacklozenge^{level_1+1}$. In the code, assuming `s` is an instance of `scale`, an expression such as `s + 1` is a convenient alternative to `scale(s.level() + 1)`. We therefore include various addition and subtraction operators in the `scale` class, all of which are applied to the `level_` member variable. The division operator still behaves according to the mathematical operation, approximating the ratio of the represented powers of 1000. Examples of these operations are shown below.

```
nano + 4      // kilo
tera - mega   // 2
micro - 3     // femto
1 + milli     // unit
milli/pico    // 1000000000
```

B.2 Duration implementation

A C++ implementation of the Appendix A.2 duration data type is outlined below. Selected `constexpr` qualifiers, and a number of member functions including comparison operators, are not shown.

```
class duration
{
public:
  constexpr duration(int64 multiplier,
                     scale precision);

  bool finite() const;
  int64 multiplier() const;
  scale precision() const;
  bool fixed() const;

  duration fixed_at(scale) const;
  duration rescaled(scale) const;
  duration refined() const;
  duration coarsened() const;
  duration unfixed() const;

  duration& operator+=(duration);
  duration& operator-=(duration);
  duration& operator*=(float64);
  duration& operator/=(float64);

  duration operator+() const;
  duration operator-() const;

  duration operator+(duration) const;
  duration operator-(duration) const;

  duration operator*(float64) const;
  duration operator/(float64) const;

  float64 operator/(duration) const;

private:
  float64 multiplier_;
  scale precision_;
  bool fixed_;
};
```

A `duration` value may be obtained using the constructor, or alternatively a user-defined literal as permitted by C++11 and subsequent standards of the programming language. The user-defined literal option requires the exact value to be known at compile time. In addition, the precision level can be no finer than `yocto` and no coarser than `yetta`. Examples of `duration` values expressed as user-defined literals are below.

```
1_s    // duration(1, unit)
1_min  // duration(60, unit)
1_hr   // duration(3600, unit)
1_day  // duration(86400, unit)
1_yr   // duration(31536000, unit)

5_ys  // duration(5, yocto)
5_zs  // duration(5, zepto)
5_as  // duration(5, atto)
5_fs  // duration(5, femto)
5_ps  // duration(5, pico)
5_ns  // duration(5, nano)
5_us  // duration(5, micro)
```

```
5_ms  // duration(5, milli)
5_ks  // duration(5, kilo)
5_Ms  // duration(5, mega)
5_Gs  // duration(5, giga)
5_Ts  // duration(5, tera)
5_Ps  // duration(5, peta)
5_Es  // duration(5, exa)
5_Zs  // duration(5, zetta)
5_Ys  // duration(5, yetta)
```

Recall from Section 3.4 the implementation strategy in which a 64-bit binary floating-point number is encapsulated in a fixed-point time class. This strategy is adopted in the `duration` class: `multiplier_` is of type `float64`, though the member functions ensure that it is rounded to an integer value as needed following every operation. As mentioned in Section 4.3, the 1000^5 limit is chosen as the largest power of 1000 less than 2^{53}, the point at which `float64` ceases to exactly represent all integers. The use of the `float64` type allows `multiplier_` to be either positive or negative infinity, supporting infinite durations. The `finite()` member function allows one to check that a `duration` value is not infinite.

Recall from Appendix A.2 that the mathematical description of duration values involves both fixed-point and base-1000 floating-point operations. These operations were denoted $+$, $-$, $\cdot$, and $/$ (fixed-point), and $\oplus$, $\ominus$, $\odot$, and $\oslash$ (floating-point). Yet the `duration` class has only one set of operators `+`, `-`, `*`, and `/`. To use these C++ operators for both fixed- and floating-point operations, we encapsulate a flag `fixed_` that determines their rounding behavior. Thus every `duration` value is either *fixed*, in which case its time precision is preserved through operations; or *unfixed*, in which case the resulting `duration` value's precision may be altered to minimize rounding error.

By default, `duration` values are unfixed, which makes it easy to express durations using combinations of multiples of base-1000 SI units. Consider the expression `3_s + 475_ms`. If the operands were fixed, this expression would raise an error since the left-hand side is in seconds whereas the right-hand side is in milliseconds. But because `duration` values are unfixed by default, `3_s + 475_ms` is equivalent to `3475_ms`.

The `fixed_at(scale)` member function maintains the expressed quantity of time, if possible, but fixes the time precision. The following expressions illustrate the effect of this operation.

```
1 4_s + 10_ms       // 4010_ms
2 (4_s + 10_ms)/4   // 1002500_us
3
4 (4_s + 10_ms).fixed_at(milli)/4
5                           // 1003_ms
6 (4_s + 10_ms).fixed_at(micro)/4
7                           // 1002500_us
```

On line 1 above, an unfixed `duration` value is created at millisecond precision. The same value is divided by 4 on line 2. Because the values are unfixed, the `duration` value resulting from the division has its time precision automatically refined to microseconds, which happens to produce the exact result. On line 4, the original expression is fixed and then divided by 4. Since the resulting `duration` value must remain in milliseconds, the result is rounded off. If the original expression is instead fixed at microseconds, as on line 6, an exact result is achieved.

The `duration` class includes a number of member functions related to precision. The `rescaled(scale)` function is almost identical to `fixed_at`, producing a `duration` value with the specified precision level. The difference is that `rescaled` neither fixes nor unfixes the time precision of the result. The member functions `refined()` and `coarsened()` are similar to `rescale`, but automatically select the finest or coarsest possible precision level that does not alter the represented duration. The `unfixed()` method produces an unfixed but otherwise equivalent `duration` value.

A key property of fixed `duration` values is that their precision levels propagate. Specifically, if an operation involves one fixed duration `fixed_dt` and one unfixed duration `unfixed_dt`, the result is generally a fixed duration with the time precision `fixed_dt.precision()`. This propagation of fixed precision levels is useful because, within an atomic model, it promotes adherence to the model's specified time quantum. Our assumption is that nearly all `duration` values produced within an atomic model instance depend in part on the instance's duration-valued parameters and on its elapsed durations. By fixing these input `duration` values at the specified time precision, any derived `duration` value will adopt the same precision level even with no conscious effort on the part of modeler.

B.3 Time point implementation

Below is an outline of a C++ `time_point` class, with comparison operators and `const` reference qualifiers omitted.

```
1  class time_point
2  {
3  public:
4    time_point();
5
6    int64 sign() const;
7    scale precision() const;
8    int64 nscales() const;
9
10   int64 scale_digit(scale) const;
11
12   time_point& advance(duration);
13
14   time_point& operator+=(duration);
15   time_point& operator-=(duration);
16
17   time_point operator+(duration) const;
18   time_point operator-(duration) const;
19
20   duration operator-(time_point) const;
21   duration gap(time_point) const;
22
23 private:
24   int8 sign_;
```

```
25    scale precision_;
26    vector<int16> digits_;
27  };
```

The class's behavior is very consistent with the mathematical operations described in Appendix A.3. Unlike the duration data type, which involved rounding in numerous operations, the rounding in the time point data type is restricted to the $\ominus$ operation implemented by the `gap(time_point)` member function. The `sign()` member function and associated member variable accommodate negative `time_point` objects. Negative time points add flexibility to the class, but greatly complicate its implementation. The `precision()` and `nscales()` functions provide, respectively, the smallest scale with a nonzero digit and the total number of stored digits. The `scale_digit(scale)` function reports the base-1000 digit at the indicated scale.

Unlike `duration`, the `time_point` class lacks multiplication and division operations. Furthermore, its computations may require memory to be dynamically allocated. These limitations reflect the fact that `time_point` is intended for simulator developers with expertise in software engineering. Modelers are expected to use the more convenient and efficient `duration` class, which may mean accepting some degree of approximation.

B.4 Time sequence implementation

The `time_sequence` class below is based on the data structure described in Appendix A.4. Algorithm 1 is implemented within the `append(time_point)` member function. The listing excludes `const` reference qualifiers and binary search operations.

```
1   class time_sequence
2   {
3   public:
4     class const_iterator;
5     using value_type =
6                   pair<int64,time_point>;
7
8     time_sequence();
9
10    bool empty() const;
11    int64 size() const;
12
13    void append(time_point);
14
15    const_iterator begin() const;
16    const_iterator end() const;
17
18    time_point front() const;
19    time_point back() const;
20    time_point operator[](int64) const;
21
22    vector<value_type> partitions() const;
23
24  private:
25    vector<value_type> partitions_;
26    vector<duration> offsets_;
27  };
```

Adhering to a recommended practice in C++, a custom iterator is used to provide access to the stored time points and support traversal. The iterator, named `time_sequence::const_iterator`, can be incremented, decremented, or offset by an arbitrary integer. It enables loops such as the one on line 11 below.

```
1   // Construct a time sequence.
2   auto ts = time_sequence();
3
4   // Populate the time sequence.
5   auto tp = time_point();
6   ts.append(tp += 5_us);
7   ts.append(tp += 72_fs);
8   ts.append(tp += 1_hr);
9
10  // Iterate through the time sequence.
11  for (const auto& entry : ts) {
12      // Obtain the current index.
13      auto i = entry.first;
14      // Obtain the current time point.
15      const auto& t = entry.second;
16      ...
17  }
```

Information stored in the iterator improves efficiency as one traverses the `time_sequence` from front (earliest time point) to back (latest time point). Accessing the time points out of order is less efficient, though still possible and convenient.

When recording simulation results, one must store not only time points but also their associated event data such as inputs, outputs, and state variables. The `time_sequence` class could be modified to retain this information. Yet our intention is that event data be recorded in a complementary structure that uses the same time point indices as the `time_sequence`. This could be interpreted as a relational approach to storing simulation results, as opposed to an object-oriented approach. Every time point in a `time_sequence` should be associated with not one event but rather a set of events, since multiple events can occur at the same point in simulated time.

B.5 Time queue implementation

The `time_queue` class implements the data structure in Appendix A.5, encapsulating Algorithms 3 though 5. Its member variables include the current time `ct_`, a vector of planned phases `queue_`, and a map `events_` which associates a set of simultaneous events with each planned phase.

```
1   class time_queue
2   {
3   public:
4       using event_set = set<int64>;
5
6       time_queue();
7
8       bool empty() const;
9       int64 size() const;
10      int64 time_count() const;
11
```

```
    time_point current_time() const;

    time_point advance();
    time_point advance(duration dt);
    time_point advance(time_point t);

    event_set imminent_events() const;
    duration imminent_duration() const;

    void pop_events();
    void pop_event(int64 event);

    void plan_event(int64 event,
                    duration dt);

    duration until(int64 event) const;
    bool cancel_event(int64 event);

private:
    time_point ct_;
    vector<duration> queue_;
    map<duration,event_set> events_;
};
```

Let us briefly describe the member functions starting from near the bottom. The `plan_event(int64, duration)` function schedules a future event using Algorithm 4 to derive the planned phase. The phase is inserted into `queue_` using a binary search based on comparison operations involving Algorithms 3 and 5.

The `until(int64)` function provides the planned duration after which the specified event will occur, whereas `cancel_event(int64)` removes the specified event. Both of these functions can be made efficient using an additional `private` member variable not shown in the above code, one that maps each event identifier to its associated planned phase and precision level.

The *imminent events* of a non-empty `time_queue` instance are those future events scheduled to occur first. The `imminent_events()` and `imminent_duration()` functions obtain these events and their associated planned duration. The `pop_event(int64)` and `pop_events()` functions may be used to remove these events when it is time for them to be processed.

Three `advance(...)` functions are provided to advance the current time `ct_`. The first, with no arguments, increases `ct_` up to the event time of the imminent events. The second, with the `dt` argument, applies multiscale time advancement. An error is produced if `dt` advances `ct_` beyond the imminent events. The third function advances the current time until it reaches the time point expressed by the `t` argument. It can be implemented by repeatedly invoking `advance(t.gap(ct_))` until `ct_` and `t` become equal.

We design the `time_queue` primarily for use in a simulator based on the DEVS formalism, which schedules at most one future event for every model instance. This means that the integer-valued event identifiers are exactly the same as the model instance identifiers. We focus specifically on Classic DEVS, for which it is a common implementation practice to order simultaneous events by the identifier of the model instance. The standard C++ `set` data structure sorts its elements by default, making it efficient to implement this simple tie-breaking mechanism. However, alternative methods for handling simultaneous events can be incorporated with minimal change to the `time_queue` class.

B.6 Time cache implementation

A simplified C++ `time_cache` implementation is shown below. Consistent with Appendix A.6, the class encapsulates an instance of `time_queue` named `tq_`.

```
class time_cache
{
public:
    time_cache();

    time_point current_time() const;

    time_point advance(duration dt);
    time_point advance(time_point t);

    void retain_event(int64 event,
                      scale precision);

    duration since(int64 event) const;
    bool release_event(int64 event);

private:
    time_queue tq_;
};
```

The two key functions are `retain_event(int64, scale)`, which applies the $1000^5 - 1$ offset and stores the resulting imaginary event in `tq_`, and `since(int64)`, which calculates an elapsed duration by subtracting from $1000^5 - 1$. The `release_event(int64)` preemptively cancels the tracking of an event, which should only be necessary if a model instance is terminated before a simulation ends.

The remaining functions pertain to the current time of the simulation. The `current_time()` function simply returns `tq_.current_time()`. The `advance(...)` functions must be invoked whenever the current time changes. These functions first remove any imaginary events in `tq_` that have been surpassed, then advance time in `tq_`.

The `time_cache` class is motivated by the prevalence of elapsed durations in DEVS-based simulations. Together, the `time_queue` and `time_cache` classes provide a general solution to event-scheduling in the presence of multiple time scales.