Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

INVESTIGATION OF VERSATILE DATATYPES FOR REPRESENTING TIME IN DISCRETE EVENT SIMULATION

Damian Vicino
Gabriel A. Wainer
Guillermo G. Trabes

Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Dr.
Ottawa, ON K1S 5B6, CANADA

ABSTRACT

Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history following a chronology of events in which the processing of each event takes place at discrete points of a continuous timeline. The simulator must interact actively with time variables for reproducing the chronology of events over positive real numbers, which is usually represented by approximated datatypes as floating-point. Nevertheless, the approximation made by commonly used datatypes in simulations can affect the timeline, preventing the generation of correct results. To overcome this problem, we present two new versatile datatypes to represent time variables in DES. These new datatypes provide a wider range of numbers reducing approximation errors, and if an error occurs, the simulation user is notified. To test our datatypes, we perform an empirical evaluation in order to compare their performance.

1 INTRODUCTION

Discrete-Event Simulation (DES) is a technique in which a simulation engine plays a history following a chronology of events. The technique is called "discrete-event" because the processing of each event of the chronology takes place at discrete points of the timeline. The events in the chronology are not needed to align to any clock; they could be placed freely over a continuous timeline, usually represented by $\mathbb{R}+$.

When implementing computer simulators, we need datatypes to represent time and its operators. In the most general case, time variables need representing arbitrary numbers in $\mathbb{R}+$. The most common approach is to choose a standard fixed-length approximated datatype such as integer or floating-point numbers.

Fixed-length approximations are adequate (and are widely adopted) for modeling and simulation (M&S) of Continuous Systems, as these methods are based on approximated results obtained for approximated input values, using approximated operations. Furthermore, the errors introduced by such approximations can be bounded (and its propagation constrained) following well-established concepts and practices from Calculus and Numerical Methods, for instance using min-max in a restricted domain (providing an interval of possible results by bounding the simulation error, using derivatives and other Calculus concepts).

In the case of DES, approximating an input can adversely affect the behavior of the model, making its trajectory diverge from that point forward. For example, an approximation could cause a permutation in the order of two events in the timeline. As we are going to show in Section 2, DES states are products of their histories; a change in their histories potentially leads to different events and eventually on errors in the results of the simulation. The datatypes adopted by most simulators, such as floating-point datatypes, approximate the numbers and are silent about the consequences the approximations can have in the result of the simulation.

In this research we present new datatypes to solve the issues mentioned. The datatypes are suitable to represent rational numbers and allow to replace floating-point or other basic datatypes. Even though the datatypes can be used on general M&S models, in this work we will focus on the time requirements and operations used by the DEVS model.

The standard way to represent the measure of a time lapse is the product between its magnitude, a time unit, and a scale-factor for the unit. The datatypes proposed in this work can represent multiple scale-factors, this way the datatypes are able to represent a wider range of numbers according to the models' requirements. This approach over the DEVS model has the characteristic that the time representation can be different on different atomic models. To provide a uniform time representation on the models, we propose a novel approach to find a common scale-factor between different number representation. Four different mechanisms to obtain a common scale-factor in coupled models are presented and evaluated. We call this search of a common scale-factor for interoperation an "agreement".

With these ideas in mind we propose: first, the datatype called Rational-Scaled Floating Point (RSFP), which provides a correct representation for all the values that can be represented with existing datatypes. Next, we present a second datatype, called Multi-Base Floating Point (MBFP) covers a different set of numbers and it has a more compact representation. The main advantages of our datatypes are two, first their ability to adjust their representation to cover a wider range of numbers, minimizing the known errors in time representation, and second, their ability to warn if any number representation issues arise, this way the user can know if the simulation gave a correct result. We also present a performance comparison using a Discrete-Event simulator based on the DEVS formalism and the DEVStone benchmark. The results obtained show how these new datatypes can even perform better than existing datatypes adding an accurate time representation.

2 RELATED WORK

Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history following a chronology of events (Wainer 2009; Zeigler et al. 2000). The technique is called "discrete-event" because the processing of each event of the chronology takes place at discrete points of the timeline. The virtual time of the simulation does not need any synchronization with real time. This allows to predict future phenomena or study complex process happening in a short time.

In the earlier days of DES, multiple M&S languages were developed, such as SIMULA (Dahl and Nygaard 1966), although according to (Nance 1981) most of these languages lacked formal soundness. The earliest approaches for formalizing DES added time semantics to well-known static modeling approaches, including well-known methods such as Timed-Automata (Alur and Dill 1994), and Generalized Semi-Markovian Processes (Glynn 1989). Other formalisms focused on concurrency problems that emerged at the time, i.e. Petri-nets (Peterson 1981), Calculus of Communicating Systems (CCS) (Milner 1982) and Communicating Sequential Processes (CSP) (Hoare 1978). We are interested in formal methods, and we will use DEVS (Zeigler et al. 2000) to conduct our research. DEVS provides a theoretical framework to think about Modeling using a hierarchical, modular approach. This formalism is proven universal for DES modeling, meaning that any model described by other DES formalism has an equivalent model in DEVS. Other characteristic of DEVS is the clear separation between model and simulation: models are described using a formal notation, and simulation algorithms are provided for running any model.

In DES, the most commonly used datatypes for representing time include floating-point, integers, fixed point, rational, and intervals between rational numbers. These datatypes are conveniently implemented in processors, or they are easily implementable combining native ones. Here, we focus on simulations that represent time as a continuous timeline, defined by $\mathbb{R}+$. The most common datatype to represent this set of numbers is floating point. Adopted by many simulators, this leads to variants due to the selection of different precision levels. For example, infinity can be mapped onto a reserved value, or an additional variable (e.g., a Boolean) can be used to handle the special case of infinity, in a structure or wrapper datatype.

The floating-point datatype was engineered to represent an approximate Real number and to support a wide range of values. The basic structure is the use of a fixed length mantissa and a fixed length exponent. The main strengths of using floating-point include:

- Compact representation, usually between 32 and 128 bits;
- Implementation in almost every processor;
- A large spectrum between the maximum and minimum number represented;
- Internal representation for infinities;
- Widely used standards make the simulator code portable;
- A mechanics of its arithmetic approximations that has been studied in detail.

On the other hand, floating-point has well known limitations (Goldberg 1991) like rounding errors, where operations (including assignment) may round the values; cancellation issues, portability (the same operation may be implemented differently in different CPUs or languages, providing different results), rounding modes, and flags/exceptions. Likewise, associativity (floating-point arithmetic is not associative, but usually used as if it was) and unexpected results for special cases: e.g. in IEEE-754, an addition overflow will result in an infinite value. These limitations are often considered an acceptable trade-off in some areas, due to their intuitive use and properties guaranteeing the errors of the result obtained could be bounded enough to produce the answers needed. In the context of DES this kind of errors can result in representation errors.

In the most generic DES formalisms, the models' domain of the time variables is $\mathbb{R}+$. This results in a quantization problem at the time of implementing the model in computers. Depending on the datatype chosen for implementation, different approximations or operation restrictions may be observed.

The most common errors in DES timelines are time shifting errors (Vicino et al. 2014). These are direct consequence of approximating the time values, reproducing events in the simulator slightly earlier or later than formally defined, but always maintaining the partial order of the events in the chronology. Time shifting errors usually have a minor impact or no impact at all on the simulation execution. For instance, in the case of normalized half-precision floating-point (16 bits) described by the IEEE-754 standard, we have a mantissa of 10 bits (plus an implicit bit in 1), an exponent of 5 bits, and a bit for the sign of the number. A number is read as $sign \cdot 2^{exp-15} \cdot mantissa$. For example the numbers 2.0039062 and 3.9980469 are represented, respectively, as: $\underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{0000000010}_{mantissa} = 1 \times 2^{16-15} \times 1.0000000010_{bin} = 2.0039062$ and $\underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{1111111111}_{mantissa} = 1 \times 2^{16-15} \times 1.1111111111_{bin} = 3.9980469$. But 2.0039062+3.9980469=6.0019531, and to represent 6.0019531 accurately we need a mantissa with more than 10 bits. The result obtained is 6, represented as $\underbrace{0}_{sign}\ \underbrace{10001}_{exponent}\ \underbrace{1000000000}_{mantissa} = 1 \times 2^{17-15} \times 1.100000000_{bin} = 6$. In logical time formalisms where only the logical order of events is considered to decide the next state (i.e. Petri Nets), this is not a problem. Nevertheless, this is a problem for timed formalisms like DEVS where it is legal to use the time elapsed since last event to define the new state, and time shifting errors may be enough to make the resulting trajectories diverge.

The second problem in time representation is the event reordering, in this case, the approximation leads to a different order of events in the timeline than the one expected using exact arithmetic on Real numbers. Here, the list of events on the timeline is permuted and the partial order of events is erroneous. In some cases, the causality chain breaks, and the resulting trajectory diverges arbitrarily from the expected one, since the following events in the chain are not necessarily related to those expected. Sometimes, an event reordering error is the result of accumulating multiple time shifting errors.

Finally, another known representation problem in time for M&S is the Zeno conditions, which occur when the definition of instantaneous or close to instantaneous actions are allowed (Lee 2014): it is possible to have infinite actions in a finite period. The distance between two events in the timeline can be defined

as any Real number, particularly those very close to zero. If the values represented are small enough, when added to the current time, they will not produce any change. This behavior can be reproduced indefinitely making the system stale. Systems producing this behavior were studied in concurrent systems (Lee 2014; Manna and Pnueli 1992). In simulation formalisms like DEVS, legitimate models never reproduce Zeno behavior (Zeigler et al. 2000). Once models' legitimacy is formally proven, the placement of events in the timeline is assumed exact. A model theoretically safe may still be unsafe in practice; a simulator using fixed-size approximated datatypes recreates the conditions for it. For example, with half-precision floating-point, adding the number $\underbrace{0}_{sign}\ \underbrace{01111}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{15-15} \times 1.0000000000_{bin} = 1$ to the number $\underbrace{0}_{sign}\ \underbrace{11011}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{27-15} \times 1.0000000000_{bin} = 2^{12} = 4096$ results in 4096, but the correct arithmetic result is 4097. Here, the result is approximated because 4097 is not that can be represented in this datatype. If this addition was to represent the time-advance of the simulation, we could reach a Zeno condition. It is important to emphasize that the Zeno problem we will focus in this work is the one that can arise in the time advance computation of the simulation in DEVS models. We will not focus on the Zeno problems on the models' internal representation, as we will assume the model is correct. The Zeno problem can easily occur in simulators using floating-point variables for representing a global clock, a common design choice in many implementations; in this case, if we simulate a metronome (a device repeatedly ticking after fixed time intervals), after enough simulation time, the accumulation in the variable renders the subsequent additions irrelevant (being too small for affecting the mantissa).

Reordering errors and Zeno problem errors are practically impossible to predict. Moreover, they may occur in irregular frequency patterns, which, in the worst case, will be undetected during verification. This was the case of the popular simulators NS-3, and OMNeT++. In (Lacage, 2010) a case was reported for NS-3 where different trajectories were obtained when using different processors to run the same simulation, due to the differences in the floating-point arithmetic implementation of each processor, including following different standards or differences in the rounding policies implemented. Having different implementations produce different approximations. A similar case was presented in (Varga and Horning 2008) about OMNeT++, the authors stated that "well-known precision problems with floating-point calculations, however, have caused problems in simulations from time to time". Neither NS-3 nor OMNeT++ investigated the problem further.

In DES, the state of the simulation can be thought as a function of its history, producing a causal relation between them. When an event in the history is approximated, the evolution of the simulation may diverge from the expected one and produce an incorrect sequence of states. When this happens, we say that the causality chain was broken. Current simulator implementations are generally silent about these errors, because usually it is impossible to detect them properly using their time datatypes. The floating-point datatypes native in most programming languages do not include any reporting of errors.

Some of the authors have done previous research in this topic. On (Vicino 2014) a preliminary version of the RSFP datatype was introduced. More recently on (Vicino 2016) a new datatype was presented to represent irrational numbers. In this paper we extend the previous works by proposing more flexible representations for rational numbers.

3 DATATYPES FOR TIME

In this section, we present new versatile datatypes for DES. We will make assumptions based in our use of the datatypes for DEVS simulators, but the concepts can be easily applied to any other DES.

The standard way to represent the measure of a time lapse is the product between its magnitude, a time unit, and a scale-factor for the unit. For example, 3 ns have a magnitude of 3, a nano scale-factor (10^{-9}) and a unit of seconds. Our proposed datatypes aim at representing magnitude and scale-factor better. As we want to represent time, the unit is implicitly assumed to be Seconds, as proposed by Bureau International de Poids et Mesures (BIPM) for International Standardization of the Time Unit. Adhering to the international standard reduces the complexity of defining conversion functions and calling them at runtime.

Our research aims to investigate more versatile datatypes that can change their representation to adapt to the values that the model may need to represent time correctly. Nevertheless, in some cases our datatypes won't be able to correctly represent the time advance, and in this case an error will be provided to the user to warn about the incorrect time representation occurred. With this information, the user can take actions to solve the problem, such as rerun the simulation with a more suited datatype. In any case, as we will show, our datatypes can represent any floating-point number and extend their range of numbers represented. This way the errors discussed before are less likely to appear.

For the datatypes proposed we cover the following functional requirements:

- Supporting multiple scale-factors.
- Providing additive operation not leading to time shifting errors, event reordering errors, or Zeno problems. - providing comparison operators (=, <).
- Including a range large enough to include every value that can be represented in floating-point and rational variables, provide safe arithmetic such that any operation with unsafe result will produce an error.
- Supporting seamlessly the coupling of models allowing the possibility of operating using multiple different scale-factors in a single simulation.

In addition, we have the following non-functional requirements: keep a compact representation, encourage optimizing the addition and comparisons over any other; and delay operations deriving from the composition of models as much as possible. These operations are known to be expensive. Based in these requirements, we propose two datatypes, Rational-Scaled floating-point (RSFP) and Multi-Base floating-point (MBFP), which are presented as replacement of floating-point and rational datatypes.

3.1 Rational Scaled Floating Point

The first datatype we propose is the Rational-Scaled floating-point (RSFP), a preliminary version of this datatype was presented on (Vicino 2014). A RSFP number is defined by four integer variables, called $quantum_n$, $quantum_d$, magnitude, and exponent. They are combined to form a number as described in Equation (1).

$$magnitude \cdot \frac{quantum_n}{quantum_d} \cdot 2^{exponent} \tag{1}$$

The unit, *second*, is implicit when using it for time. The *exponent* and *quantum* variables take care of scaling the units when needed. We call this group of variables the *scale-factor*. The quotient between $quantum_n$ and $quantum_d$ we call it the ratio. For example, a time lapse of 3 nanoseconds can be expressed in the following way: $3 \cdot \frac{1}{5^9} \cdot 2^{-9} \cdot$ s, where the *magnitude* is 3, the *scale-factor* is 10^{-9} and the unit is *second*. This new datatype provides a correct representation for all the numbers implemented in existing simulators. For example, a floating-point value can be converted to a RSFP setting both *quantum* variables to one matching the *exponents* and assigning the *mantissa* to the *magnitude*. Depending on the unit associated to the original value, some minor adjustments to the *quantum* and *exponent* variables may be needed. For example, the binary floating-point 1E-3 $\frac{1}{8}$ milliseconds can be defined in RSFP with $quantum_n$ = 1, $quantum_d$= 1000, *magnitude* = 1, and *exponent* = -3. This equivalence derivation is as follows: $\frac{1}{8} \cdot ms = 1 \cdot 2^{-3} \cdot ms = \frac{1}{1000} \cdot 2^{-3} \cdot s = 1 \cdot \frac{1}{1000} \cdot 2^{-3} \cdot$ s. In addition, RSFP can be used to represent and operate in a large range, covering all previously available representation values (floating-point and rational) safely, and introducing new numbers not available on any of the previous datatypes. For example, $0.1 \cdot 2^{128}$ can be represented in RSFP, but

it has infinite binary periodicity, and therefore it cannot be represented using binary floating-point datatypes, and it is too large to be represented as rational.

3.2 Multi-Based Floating Point

The RSFP datatype, presented in the previous section, solves basic issues in current simulators; nevertheless, it has usability limitations. Our first concern is a representation restriction over large and small numbers. We can represent any (binary) floating-point number. Nevertheless, we cannot represent any large or small number produced by multiplying only co-primes of 2. For example, the largest odd number being represented must use exponent 0 (or its simplification against $quantum_d$ be 0), because using any larger value makes the number even. Then, an accumulator repeatedly adding an odd number reaches a point where next addition result is not that can be represented in RSFP. This is a consequence of not having addition-closure in the datatype.

To solve these issues, we propose the Multi-Base floating-point datatype (MBFP) as an alternative to RSFP. This datatype covers a different set of numbers, and it has a more compact representation. These characteristics make MBFP more appropriate than RSFP in some simulation scenarios. In Equation (2) we show how to interpret a number represented using MBFP.

$$mantissa \cdot base^{exponent} \tag{2}$$

The MBFP datatype can be implemented using three integer variables representing *base*, *mantissa*, and *exponent*. For example, 3^{-70} can be represented in MBFP setting *mantissa* to 1, *base* to 3, and *exponent* to -70. Operations are simpler, because only the *base* needs agreements. A drawback here is the need to work with non-binary powers, which are not optimized the same as powers of 2. However, never need to use compute power operations using high exponents. We are limited to use exponents below the size of the *mantissa*. Multiplying for a power of the base is seen as shifting digits (left or right) in the *mantissa* (considering the base in use).

This new datatype also provides a correct representation for all those numbers represented in current datatypes for DES. For converting an integer to MBFP, we set the integer value in the *mantissa*. If the integer was representing seconds, we set the *exponent* to 0, and the *base* to 2. Otherwise, we need to adjust the unit. For example, for milliseconds, we set *base* to 10, and *exponent* to -3. Every integer with equal or smaller size than the *mantissa* can be represented using this conversion. Conversion from floating-point is also simple. We set the *base* to 2 and copy the *exponent* and *mantissa*. We need to add the implicit 1 when writing the MBFP *mantissa*. An example of a half-precision floating-point converted to MBFP is $\underbrace{0}_{sign}\ \underbrace{10001}_{exponent}\ \underbrace{1100100000}_{mantissa} = 1.1100100000_{bin} \times 2^{17-15} = 1.78125 \times 4 = 7.12 = 2^{-5} \times 57 \times 2^{2} = 57 \times 2^{-3} = MBFP: <\underbrace{57}_{mantissa}, \underbrace{2}_{base}, \underbrace{-3}_{exponent}>$. Conversion from Rational can be done by setting the *exponent* to -1, the denominator as the *base* and the numerator as the *mantissa*. Conversion from and to RSFP is not always possible, even assuming both use same integer size for internal variables. An example of a valid RSFP conversion to MBFP is $RSFP: <\underbrace{3}_{quantum_n}, \underbrace{5^5}_{quantum_d}, \underbrace{1000}_{magnitude}, \underbrace{5}_{exponent}> \rightarrow \frac{3000}{5^5} \times 2^5 = \frac{3}{3125} \times 1000 \times 2^5 = 3000 \times 4^5 \times 10^{-5} = 3072000 \times 10^{-5} \rightarrow MBFP: <\underbrace{3072}_{mantissa}, \underbrace{10}_{base}, \underbrace{-2}_{exponent}>$. If the RSFP converted has a large exponent, the denominator cannot be defined as a base with a negative exponent, making it impossible to convert to MBFP if $quantum_d$ is not zero. An example of RSFP that cannot be converted to MBFP is $RSFP: <\underbrace{1}_{quantum_n}, \underbrace{5}_{quantum_d}, \underbrace{1}_{magnitude}, \underbrace{2^{20}}_{exponent}> \rightarrow \frac{1}{5} \times 1 \times 2^{1048576} = 5^{-1} \times 2^{1048576} \rightarrow MBFP: ?$. In this example, if we set the *base* to 5 and the *exponent* to -1, the product of the *magnitude* by the power does not fit in the *mantissa*, and if we set the *base* to 2, we are not allowed to set a denominator. Finally, some numbers represented in

MBFP, as 3^{-70}, cannot be converted to RSFP. This shows that MBFP and RSFP representation domains overlap, but they are not equal. In different application domains we may find a better fit for each of them.

3.3 Balancing the Representation

An important characteristic of our proposed datatypes is that they have redundant values, like in the rational representation. We can exploit this redundancy to work around operation overflows of any component of the datatype. For example, to represent a second on RSFP, we can do it as: $1 \cdot \frac{1}{1} \cdot 2^0 \cdot \text{s} = 2 \cdot \frac{1}{2} \cdot 2^0 \cdot \text{s} = 1 \cdot \frac{1}{2} \cdot 2^1 \cdot \text{s}$.

For the RSFP datatype, if an overflow on $quantum_n$ occurs, we balance with $quantum_d$. If simplifying is not enough, we can transfer weight to the magnitude, dividing $quantum_n$ by a divisor and multiply the magnitude by the same divisor. Another alternative, if the binary representation ends in 0, magnitude digits can be shifted, and the exponent adjusted to compensate. For example, using 32-bit integers, if the result of an operation produces $<2^{33}, 8, 1, 4>$, we cannot fit the value of $quantum_n$. Instead, we can increment the exponent to 10 and set $quantum_n$ to 2^{27} and it fits. Or we can simplify $quantum_n$ against $quantum_d$, setting $quantum_n$ to 2^{30} and $quantum_d$ to 2^3. Or we can increment the *magnitude*, for instance to 16, and reduce $quantum_n$ to 2^{29}. Prioritizing any of these tactics over others is an implementation detail. For overflow on $quantum_d$, simplification against $quantum_n$, magnitude, or exponent can be used. We can simplify $quantum_d$ against the magnitude. For example, using 32-bit integers, if the result of an operation produces $<8, 2^{33}, 4, 0>$, we cannot fit the value of $quantum_d$ in its variable. Instead, we can decrement the exponent to -10 and set $quantum_d$ to 2^{23}. Or, we can simplify $quantum_d$ against $quantum_n$, setting $quantum_n$ to 1 and $quantum_d$ to 2^{30}. Or, we can simplify $quantum_d$ against the magnitude, setting magnitude to 1 and $quantum_d$ to 2^{31}. Prioritizing any of these tactics over others is an implementation detail. For an overflow on the magnitude, we can simplify it against $quantum_d$. Or, we can transfer some weight to $quantum_n$. In some cases, we can shift digits and adjust the exponent. The options for $quantum_n$ and *magnitude* are similar, since numerically permuting $quantum_n$ and *magnitude* have the same meaning. The difference between these variables is the intended use, which impacts in the design of the operations. For example, using 32-bit integers, if the result of an operation produces a $quantum_n$ of 1, a $quantum_d$ of 8, a *magnitude* of 2^{33}, and an *exponent* of 4, we cannot fit the value of magnitude in its variable. We can increment the *exponent* to 10 and set the *magnitude* to 2^{27}. We can simplify against $quantum_d$, setting magnitude to 2^{30} and $quantum_d$ to 64; or we can increment the $quantum_n$ to some value as 16 and reduce the same factor from magnitude setting it to 2^{29}. All these manipulations of internal representation should be delayed until they are needed to keep complexity low. Running a simplification needs computing expensive algorithms to run as Greater Common Divisor (GCD) and Least Common Multiple (LCM).

A similar analysis can be done for the MBFP datatype, where the any overflow over the *mantissa, base* or *exponent* can be handled adjusting the parameters.

3.4 Agreeing on a Scale-Factor for Representation

Our proposed datatypes can adapt the representation to cover different sets of numbers needed by the models. This approach has the drawback that the representation can be different on DEVS atomic models that must interact. For the design of our datatypes, we will set every time advance in a model with an identical scale-factor. This assumption is a relaxed version of the one made by most current simulators, in which every model normally uses the same time unit and scale-factor. Under our assumption, we must find a scale-factor value for all the coupled models connecting atomic components. We call this search of a common scale-factor for interoperation an "agreement". Having mechanisms to discover a common scale-factor for a coupled model, allows us to iterate and find a common scale-factor for the simulation. This globally common scale-factor allow us to reduce the complexity of our simulation.

On the RSFP datatype for finding an agreement between two numbers A and B, and agreement on the ratio and exponent must be found. To achieve this, first a common denominator is found with the LCM of

the $quantum_d$ values from A and B. The exponent is the minimal between A and B, then the $quantum_n$ and *magnitude* should be adjusted. An example of an agreement for addition between two RSFP numbers is showed next. $A: <\underbrace{3}_{quantum_n}, \underbrace{5}_{quantum_d}, \underbrace{1}_{magnitude}, \underbrace{5}_{exponent}>$, $B: <\underbrace{5}_{quantum_n}, \underbrace{10}_{quantum_d}, \underbrace{2}_{magnitude}, \underbrace{6}_{exponent}>$, $LCM\,(5,10) = 10$, $\min(5,6) = 5$, After Adjustment: $A: <\underbrace{1}_{quantum_n}, \underbrace{10}_{quantum_d}, \underbrace{6}_{magnitude}, \underbrace{5}_{exponent}>$, $B: <\underbrace{1}_{quantum_n}, \underbrace{10}_{quantum_d}, \underbrace{20}_{magnitude}, \underbrace{5}_{exponent}>$, $Result: \mathrm{A} + \mathrm{B} = <\underbrace{1}_{quantum_n}, \underbrace{10}_{quantum_d}, \underbrace{26}_{magnitude}, \underbrace{5}_{exponent}>$.

On the MBFP datatype the rule for agreement between two numbers A and B is the following: the common base is the LCM of the bases of A and B, the exponent is the minimal exponent between A and B, and the mantissa needs to be adjusted for compensation shifting digits. Next, we show an example of an agreement for addition between two MBFP.

$A: <\underbrace{2}_{mantissa}, \underbrace{3}_{base}, \underbrace{-2}_{exponent}>$, $B: <\underbrace{4}_{mantissa}, \underbrace{4}_{base}, \underbrace{2}_{exponent}>$, $LCM\,(3,4) = 12$, $\min(-2,2) = -2$

$After\ adjustment: \underbrace{32 \times 12^{-2}}_{A} + \underbrace{11520 \times 12^{-2}}_{B} = \underbrace{11552 \times 12^{-2}}_{Result}$, $Result: <\underbrace{11520}_{mantissa}, \underbrace{12}_{base}, \underbrace{-2}_{exponent}>$.

To reach a globally common *scale-factor* agreement on a DEVS model several approaches can be implemented. We propose four approaches the two datatypes proposed: static, dynamic-initialization, local, and global.

In the static approaches, called RSFP-static and MBFP-static, the modeler needs to declare the scale-factor in advance for each atomic model, and then a preprocessor or compiler computes the common scale-factor. After finding it, every value in the models is adjusted. Before the simulation starts, we define a variable for each scale-factor, and we call the establish-static-scale-factor function based on template parameters. This approach can be implemented only in languages that allow processing operations at compile time, like C++.

In the dynamic-initialization approach, called RSFP-init and MBFP-init, each variable internally keeps a scale-factor. We must declare at least one variable of each scale-factor to be used. Each variable is registered in a global queue. After declaring all the scale-factors, and before the simulation starts, an agreement is negotiated, and every variable in the queue is adjusted. Like the static approach, it is possible to relax our assumption of single scale-factor per model as far as every scale-factor is declared before the simulation starts. Like in the static approach, we still need to declare scale-factors in advance, and there is no straightforward way to prevent the use of balancing the representation, adding overhead. Unlike the static approach, it can be implemented in languages like Python or PHP, which do not support the evaluation of expressions at compile time.

In the local approaches, called RSFP-local and MBFP-local, we find agreements between parameters of binary operations. A global agreement may never be reached, but that does not affect the results of the simulation. Interaction between models is the driving force of the propagation. Thus, models with low level or no interaction may create clusters of scale-factors (i.e., a group of variables sharing a common scale-factor). As we use a single scale-factor per model, after the models interact with them each will reach a stable factor. We do not need casting operations or calling to a common established scale-factor, as each variable has its own. For binary operations, the scale-factors of operands are checked and adjusted to a common scale-factor if needed.

Finally, in the global approaches, called RSFP-global and MBFP-global, a globally accessible scale-factor variable is defined. For each operation, each operand scale-factor is compared with the global. In case it does not match the global, an agreement is searched and registered. The agreement eventually reaches a global consensus. In many cases, the global approach converges faster than the local one, but it has limitations for concurrency. Using global variables may not be suitable for distributed simulators, and it is a bottleneck in multi-threading. The global approach gets to a scale-factor representation agreement faster because it only needs setting each model scale-factor representation to the global once. After this, to propagate the scale-factor, only one operation with each variable is needed. In contrast, in the local approach

the agreement is driven by the operations, and a chain of operations between local agreements is needed to obtain higher level agreements and a chain of binary operations involving every variable is needed for propagating the representation.

For the local and global approaches, we do not need to declare scale-factors in advance. Their performance penalty is only significant until the scale-factors of the simulation stabilizes. In the worst-case scenario, every operation produces a penalty for computing Greatest Common Divisor (GCD) and Lowest Common Multiple (LCM) during the agreement. However, under our assumption of a single scale-factor per model, finding agreements is easier, because each GCD and LCM work as accumulators of information. If we find an agreement *c* between A and B, the agreements between *c* and A, *c* and B, are again *c*.

3.5 Operations

The abstract DEVS simulator presented in (Zeigler et al, 2000), which we use, only operate on time variables through four operations: addition, subtraction, equality comparison, and "lower than" comparison: comparison for deciding which transition to execute next, addition for advancing the chronology, and subtraction for obtaining the elapsed time.

To compare if two RSFP datatype numbers are equal, checking the variables is enough to provide a positive answer. If any of the variables differ, we cannot decide, then we produce an agreement for the scale-factor and compare their magnitudes. If there is no agreement, we know that the numbers are not equal. Else, we compare their magnitudes. In contrast, to compare if two MBFP datatypes are equal, after finding an agreement for the bases, the exponent of both operands must be matched. The difference between exponents is compensated by shifting digits in the *mantissa*. Then the mantissas can be compared to obtain an answer.

On RSFP when comparing by lower-than, several things need to be checked.

- Check if the operands are equal. If they are, we can give a negative answer.
- If not, one of them is lower than the other. If an agreement can be found, we compare the magnitudes.
- If an agreement cannot be found (i.e., the difference between two operands is large), we can still compute the comparison by lower than.
- We check if the signs of both numbers match; otherwise the negative is lower than the positive.
- We search for a partial agreement by adjusting the operands to have the same $quantum_d$, and the closest possible exponents without rounding. Having a common $quantum_d$, we can compare numbers looking at their exponents' difference. If the difference between their exponents is larger than two times the size of the integers used to represent the internal values, then the large exponent correspond to larger value in positive numbers, and lower exponent corresponds to larger value in negative numbers.
- If not, we can use a temporary larger datatype for comparing the *magnitude*, $quantum_n$ and *exponents*. In this case, a common scale-factor is not produced, and the *exponent* difference is not enough to decide using the common $quantum_d$. Then, we must compare $quantum_n$ and magnitude values too. On MBFP when compare by lower-than, the algorithm is like the one for comparison, given than an agreement for the *base* was made, the exponents of both operands must be matched. After these steps comparing the mantissas will give a result.

Finally, for RSFP, the addition operation usually adds the magnitudes of numbers in agreement. A special case is overflow. In this case we can return a result that is not represented in the same scale-factor or raise an error. The introduction of a new scale-factor may cascade in a global readjustment of variables. In case the scale-factor of the operands have different exponent, we first check their difference. If the difference is larger than two times the bits of the magnitude, we raise an error. Otherwise, we use a temporary variable to compute the addition using the lower exponent in both variables and balance the result. If the

result does not fit the original variable after balancing, an error is raised. On MBFP the addition can be made between numbers in agreement by matching their exponents. The difference between exponents is compensated by shifting digits in the *mantissa*. If we must shift more digits than the size of the *mantissa*, the mantissa losses values. In this case a balance on the representation can be made and if it is not possible an error is notified.

3.6 Handling Errors

Comparison operators always return a result, but the addition may throw an error. This occurs when we fail to reach an agreement on the result's *scale-factor*, because our datatype does not have addition-closure. Nevertheless, this problem happens less in RSFP and MBFP than in floating-point, because the timeline represented by our datatypes is denser, and balancing can prevent it. If the issue arises, we detect and notify the error, as opposed to floating-point that silently approximates and continues.

In the case an error is notified, we have options to complete the simulation. First, we can increase the representation of our datatypes by using a larger datatype on any of the parameters. If a proper representation cannot be estimated, or the programming language does not allow it, a dynamic arbitrary length can be used, i.e. the BigInt provided by the Boost multi-precision library. We do not use this kind of datatype by default because of their low performance. A second alternative would be to introduce approximation algorithms to run when the error is detected. This would allow the simulation to incorrectly advance the trajectory, but it should still mark in the timeline where and what errors were introduced for after-run analysis.

4 EXPERIMENTAL EVALUATION AND RESULTS

Predicting when representation errors will appear is not an easy task, and we propose to study analytically and empirically this problem as future work. In this paper we will focus on an empirical evaluation of how our datatypes perform. The main drawback of our datatypes is that a more complex representation can lead to a more expensive computation time to execute a model. Also, the agreement for the scale-factor needed by our datatypes can create an additional overhead.

To evaluate the performance of our proposed datatypes empirically, we implemented every proposed datatype in C++14 and compared them against the native datatype double. We used the DEVStone synthetic benchmark (Wainer et al. 2011) implemented for a customized version of aDEVS-2.8. DEVStone helps automating the evaluation of DEVS based simulators, and it can be adapted to other DES engines. It generates a suite of models of different sizes, complexities and behaviors like diverse applications that exist in the real world. Different types of models with different internal and external structure can be used. For this work we will use the following ones:

- LI: Models with a low level of interconnections for each coupled model. Each coupled component has only one input and one output port. The input port is connected to each component but only one component produced an output through the output port.
- HI: Models with a high level of input couplings. HI models have the same number of atomic components with more interconnections: each atomic component (a) connects its output port to the input port of the (a+1)th component.

All experiments were compiled using clang 4.1 over the x86-64 version of FreeBSD 10.1-18 operating system. The compiler was used with level 2 optimization. To compare the different implementations, the four *scale-factor* agreement techniques discussed in Section 3.1.1 were used on the experiments (static, init, local and global) for the RSFP and for the MBFP datatypes.

Table 1 shows the results obtained for each datatype in four Low-level of Interconnections (LI) configuration experiments, and in four High-level of Input coupling (HI) configuration experiments. To simplify comparisons, the overhead is measured relatively to the one obtained for double. In these experiments, the

period of time-advance used by the DEVStone models was handpicked to produce correct results using any datatypes being compared. Every experiment was executed 20 times, with no significant variance in the results. In Table 2, we show the theoretical number of transitions being executed according to the DEVStone formulas to show how our experimentation was performed over a representative set of different models but with a similar amount of internal and external transitions.

From the obtained results, we see RSFP-static and MBFP-local are leading the performance comparison. And, they produce even better results on LI than HI. We attribute this to the fact that HI configurations make more intensive use of comparison than arithmetic operations. The worst performance obtained in the LI configurations is MBFP-static. This is attributed to power function used for adjusting the mantissa when the exponents are different in the operands. This is not a problem for RSFP-static because it removes any adjustment by fixing the scale-factor at compile time. The difference with others in the MBFP family is due to our implementation using C++ templates preventing some optimization in the compiler. This is a point that should be looked further to fully understand its implications.

Table 1: DEVStone comparing time datatypes in aDEVS.

	LI											
datatype	width	depth	Overhead	width	depth	overhead	width	depth	overhead	width	depth	overhead
RSFP-static			-12.35 %			-10.77 %			-11.33 %			-7.48 %
RSFP-init			12.55 %			12.98 %			19.40 %			17.25 %
RSFP-global			19.56 %			-19.56 %			19.56 %			19.56 %
RSFP-local			-4.90 %			-7.08 %			-6.80 %			-5.87 %
double	51	10001	0.00 %	501	1001	0.00 %	5001	101	0.00 %	50001	11	0.00 %
MBFP-static			37.99 %			73.23 %			81.39 %			78.97 %
MBFP-init			6.62 %			15.09 %			18.36 %			20.08 %
MBFP-global			3.95 %			7.58 %			6.95 %			8.32 %
MBFP-local			-7.39 %			-9.43 %			-9.97 %			-7.34 %
	HI											
datatype	width	depth	overhead	width	depth	overhead	width	depth	overhead	width	depth	overhead
RSFP static			-27.89 %			-21.81 %			-36.67 %			-42.77 %
RSFP-init			11.28 %			31.28 %			-1.12 %			12.71 %
RSFP-global			-1.06 %			8.67 %			-11.71 %			-13.00 %
RSFP-local			-25.17 %			-17.20 %			-28.91 %			-21.14 %
double	100	200	0.00 %	142	100	0.00 %	205	50	0.00 %	720	5	0.00 %
MBFP-static			-16.29 %			-10.82 %			-28.42 %			6.15 %
MBFP-init			-3.11 %			4.24 %			-18.54 %			23.50 %
MBFP-global			-15.74 %			-3.91 %			-16.01 %			16.35 %
MBFP-local			-30.61 %			-17.89 %			-34.46 %			-40.47 %

Table 2: DEVStone theoretical transitions on experiments.

configuration	width	depth	$\#\delta_{int}$ transitions	$\#\delta_{ext}$ transitions	configuration	width	depth	$\#\delta_{int}$ transitions	$\#\delta_{ext}$ transitions
LI	51	10001	499999	499999	HI	100	200	1004651	1004651
	501	1001	499999	499999		142	100	1005148	1005148
	5001	101	499999	499999		205	50	1034636	1034636
	50001	11	499999	499999		720	5	1038241	1038241

In both datatypes, RSFP and MBFP, the init and global implementations are doing worse than others in performance. We attribute this to the loss of locality affecting the cache. We believe in the value of these datatypes for context were CPU have lower sizes of cache.

5 CONCLUSIONS AND FUTURE WORK

In this work we presented the problems that arise in time representation in DES in current simulators. To solve these problems two new datatypes were presented. We also presented four different mechanisms to assure a globally common scale-factor on a DEVS model. We discussed these datatypes limitations and presented an empirical evaluation of their performance using the DEVStone benchmark. The results obtained from the empirical evaluation show how these new types can be used in simulations without creating

significant overhead. As future work we propose to extend the work with mathematical analysis of the datatypes and how they solve the representation problems and to evaluate these datatypes on different platforms.

REFERENCES

Alur, R. and D. L. Dill. 1994. "A Theory of Timed Automata". *Theoretical Computer Science* 126(2):183–235.

Bolduc, J.-S. and H. Vangheluwe. 2002. "The Modelling and Simulation Package pythonDEVS for Classical Hierarchical DEVS". MSDL Technical Report MSDL-TR-2001-01, McGill University, Montreal, QC, Canada.

Dahl, O.-J. and K. Nygaard. 1966. "SIMULA: An ALGOL-based Simulation Language". *Communications of ACM.* 9(9):671–678.

Goldberg, D. 1991. "What Every Computer Scientist Should Know About Floating-point Arithmetic". *ACM Computing Surveys* 23(1):5–48.

Glynn, P. W. 1989. "A GSMP Formalism for Discrete Event Systems". *Proceedings of the IEEE* 77(1):14–23.

Himmelspach, J. and A. M. Uhrmacher. 2007. "Plug'N Simulate". In *ANNS '07: Proceedings of the 40th Annual Simulation Symposium,* edited by H. Karatza and T. F. Znati, 137–143, Washington, DC, USA: IEE Computer Society.

Hoare, C. A. R. 1978. "Communicating Sequential Processes". *Communications of ACM* 21(8):666–677.

Hwang, M. H. 2007. "DEVS++: C++ Open Source Library of DEVS Formalism". http://odevspp.sourceforge.net/, accessed 28th June 2019.

Janousek, V. and E. Kironsky. 2006. "Exploratory Modeling with SmallDEVS". In *Proceedings of the 20th annual European Simulations and Modelling Conference*, edited by A. Nkesta, M. Paludetto and C. Bertelle, 122-126, Ghent, Belgium: EUROSIS-ETI.

Lacage, M. 2010. *Experimentation Tools for Networking Research*. Ph.D. thesis, Universite De Nice-Sophia Antipolis, France.

Lee, E. A. 2014. "Constructive Models of Discrete and Continuous Physical Phenomena". *Technical Report UCB/EECS-2014-15,* EECS Department, University of California, Berkeley.

Manna, Z. and A. Pnueli. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Berlin, Heidelberg: Springer-Verlag.

Milner, R. 1982. *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag.

Nance, R. E. 1981. "The Time and State Relationships in Simulation Modeling". *Communications of ACM* 24(4):173–179.

Peterson, J. L. 1981. *Petri Net Theory and the Modeling of Systems*. Prentice Hall Inc., Englewood Cliffs, New York, USA.

Varga, A. and R. Hornig. 2008. "An Overview of the OMNeT++ Simulation Environment". In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems,* edited by O. Dalle and G. Wainer, Article 60, Brussels, Belgium: ICST.

Vicino, D., O. Dalle, and G. A. Wainer. 2014. "A Data Type for Discretized Time Representation in DEVS". In *Proceedings of 7th International ICST Conference on Simulation Tools and Techniques*, edited by F. Barros, K. Perumalla and R. Ewald, 11-20, Brussels, Belgium: ICST.

Vicino, D., O. Dalle, and G. A. Wainer. 2016. "An Advanced Data Type with Irrational Numbers to Implement Time in DEVS Simulators". In *Proceedings of TMS/DEVS Symposium on Theory of Modeling & Simulation*, edited by F. Barros, X. Hu, H. Prahofer and J. Denil, 164-171, Red Hook, NY, USA: Curran Associates Inc.

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL, USA, CRC Press, Inc..

Wainer, G. A., E. Glinsky, and M. Gutierrez-Alcaraz. 2011. "Studying Performance of DEVS Modeling and Simulation Environments Using the DEVStone Benchmark". *Simulation*, 87(7):555–580.

Zeigler, B. P., H. Praehofer, and T. Kim. 2000. *Theory of Modeling and Simulation* (2nd ed.). Orlando, FL, USA: Academic Press, Inc.

Zeigler, B. P. and H. Sarjoughian. 2003. *Introduction to DEVS Modeling and Simulation with Java: Developing Component-based Simulation Models*. Tempe, Arizona, USA: Arizona State University.

AUTHOR BIOGRAPHIES

DAMIAN VICINO has obtained a co-joint Ph.D. in Computer Science (Université de Nice-Sophia Antipolis) and Systems and Computer Engineering (Carleton University). Currently, he is a Software Developer Engineer at Amazon working in Alexa product. His email address is damianvicino@cmail.carleton.ca. Note: this research was done prior to joining Amazon.

GABRIEL WAINER is Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of the Society for Modeling and Simulation International (SCS). His email address is gwainer@sce.carleton.ca.

GUILLERMO TRABES is a Ph.D. student in Electrical and Computer Engineering (Carleton University) and Computer Science (Universidad Nacional de San Luis). His email address is guillermotrabes@sce.carleton.ca.