Journal of Computational Science 5 (2014) 882–890

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

On the definition of a computational fluid dynamic solver using cellular discrete-event simulation

Michael Van Schyndel[a], Gabriel A. Wainer[a,*], Rhys Goldstein[b], Jeremy P.M. Mogk[b], Azam Khan[b]

[a] Department of Systems & Computer Engineering, Carleton University, Center for Visualization and Simulation (VSIM), Ottawa, ON, Canada
[b] Autodesk Research Toronto, ON, Canada

ARTICLE INFO

Article history:
Received 30 October 2013
Received in revised form 1 May 2014
Accepted 1 June 2014
Available online 9 June 2014

Keywords:
Computational fluid dynamics
Cellular Automata
Discrete event system
Biomechanical simulations

ABSTRACT

The Discrete Event System Specification (DEVS) has rarely been applied to the physics of motion. To explore the formalism's potential contribution to these applications, we need to investigate the definition of moving gases, liquids, rigid bodies, and deformable solids. Here, we show how to use Cell-DEVS to analyze the movement and interactions of fluids using computational fluid dynamics (CFD). We describe a set of rules that produce the same patterns as traditional CFD implementations. We present the inner workings of the CFD algorithm, the incorporation of solid barriers, and the adoption of variable time steps within the context of biomechanical simulations.

1. Introduction and motivation

The Discrete Event System Specification (DEVS) formalism, described in [1], has two properties that facilitate a scalable approach to simulation development. First, DEVS has been shown to be exceptionally general when compared with other modeling formalisms [2], allowing it to be applied to a wide range of simulation methods. Second, multiple DEVS models are easily coupled to represent more complex systems, even if the component models exhibit different time advancement patterns. A key principle of DEVS is that a model, the computer code which pertains to a specific real-world system, is separated from the simulator, the computer program which advances time.

Although there are numerous applications of DEVS to artificial systems, environmental systems, and both biological and physical processes, the formalism is rarely applied to the physics of motion in two or three spatial dimensions. Simulations involving moving gases, liquids, rigid bodies, deformable solids, or a mixture of substances are usually implemented using traditional simulation techniques. Moreover, the timestep is generally fixed, and there is typically no separation between model and simulator. The primary rationale for DEVS-based models of 2D and 3D solids and fluids in motion is the ease with which DEVS-based models can be coupled with one another. For example, researchers and engineers in the field of biomechanics could simulate an implanted medical device by coupling a DEVS model of the device with another DEVS model of the surrounding tissue. Another benefit of DEVS is the formalism's support of different time advancement patterns. For example, the medical device model might be based on an event-driven approach, whereas the tissue model might use either fixed time steps or time steps that shorten in response to fast motion.

Here we apply DEVS to Computational Fluid Dynamics (CFD), the numerical methods and algorithms which solve and analyze the movement and interactions of fluid flows [3]. In general, no analytical solution exists for non-linear fluid models; hence, the numerical approximation methods, also called "computational models," become important. One promising theoretical approach toward resolving CFD-specific problems is the adoption of discrete-event methodologies. CFD Solvers are required to process problems comprised of a large number of computations, which makes the use of computer-based approaches inevitable. In computerized processing of CFD, a boundary for the problem is defined, and the environment is divided into a cellular space in which each cell represents a physical volume. Motion within the defined environment evolves in accordance with the fundamental principles of mass, momentum, and energy conservation. The behavior of the fluid at the boundaries is also defined, termed the boundary

* Corresponding author. Tel.: +1 6135202600.
E-mail addresses: mvschynd@connect.carleton.ca (M. Van Schyndel), gwainer@sce.carleton.ca (G.A. Wainer), rhys.goldstein@autodesk.com (R. Goldstein), jeremy.mogk@autodesk.com (J.P.M. Mogk), azam.khan@autodesk.com (A. Khan).

http://dx.doi.org/10.1016/j.jocs.2014.06.001
1877-7503/

conditions. These specifications construct a model of the fluid which can then be simulated on a powerful computing device. The simulation solves the problem by computing the equations of each cell during a specific duration of time. Finally, visualization and analysis of the results can render a meaningful and sensible outcome of the computations.

Different cellular methods have been proposed to solve these problems. In particular, Cellular Automata (CA) theory [4] is a branch of discrete dynamic systems, in which space is represented by a cellular grid, where each cell is a state machine. In CA, the time advances in a discrete manner and triggers state changes in the cells based on the value of their neighbor cells. This theory has been used in physics, complexity science, theoretical biology, microstructure modeling, and spatial modeling. The Cell-DEVS (Cellular Discrete Event System Specification) formalism [1,5] is a related formalism in which each cell evolves asynchronously using explicit timing delays. This solves the problem of unnecessary processing burden in cells and allows for more efficient asynchronous execution using a continuous time-base, without losing accuracy. In this methodology, each cell is represented as a DEVS atomic model that changes state in response to the occurrence of events in an event-driven fashion.

Cell-DEVS was originally introduced for modeling and simulation of spatial systems; however, not until the current research has anyone proposed using the Cell-DEVS methodology to implement physics-based CFD equations to simulate fluid dynamics. The rule-based nature of defining cellular model behavior provides a platform to define area-wise behavior, leading to easier and faster adoption and implementation of CFD solver algorithms. The other advantage of this method is its fast computing apparatus that works asynchronously on the cellular grid, thus increasing the execution speed. The continuous time-advance nature of Cell-DEVS can contribute to the seamless simulation of CFD, in comparison with the discrete timing in CA that lacks the smoothness of fluid flow. Cell-DEVS models are able to generate realistic results with reasonable speed. Finally, the formal I/O port definitions permit output signals to be produced based on satisfying a specific condition in the cell lattice, and allows the transfer of data between different spatial components. Furthermore, the solver simulation can be interfaced with advanced visualization software to provide a realistic graphical display [6,7].

The solver proposed here uses the DEVS formalism to enable the solution and analysis of fluid flow behaviors using a set of simple and stable CFD algorithms. We describe the Cell-DEVS implementation of these algorithms, supplemented by sample CD++ code, and present simulation results to demonstrate the feasibility of the approach [8]. This is the first successful attempt in modeling CFD as a discrete-event systems specification and we focus on how those results were achieved. Finally, we outline how to address the development of increasingly complex models, including the incorporation of solid barriers and the execution of variable time steps, illustrating its importance within the context of biomechanical and biomedical applications.

2. Related work

Fluid dynamic solvers are used for a wide variety of purposes. Their goal is to create a realistic representation of a naturally occurring fluid system such as rising smoke or blowing dust. The flow of fluids can be viewed as solid particles interacting with velocity fields or as densities. There are different methods for predicting the evolution of these fields and densities: the lattice-gas [9], Navier–Stokes equations [10] and Riemann Solvers [11].

In general, CFD methods are categorized into two groups; (i) Discretization methods and (ii) Turbulence models. Discretization methods are a subset of the divide and conquer approach for solving difficult computational problems, in which the computational domain is discretized and "each term within the partial differential equation describing the flow is written in such a manner that the computer can be programmed to calculate" [12]. Turbulence models are designed to address the unsteady motions that can affect flow, but cannot be directly resolved. The choice of model is typically dictated by the form of the governing equations that were applied, which often relates to the context of the simulation. The model is used to generate solutions at a variety of length and time scales; the more scales that are resolved, the more detailed the flow patterns.

Navier–Stokes equations were the first physical description of fluid motion, a set of differential equations derived from the laws of classical dynamics. The first comprehensive simulation of the Navier–Stokes equations appeared in 1986 [13], and demonstrated that detail to the level of real molecular dynamics was not necessary to cause realistic fluid mechanics. In the book by Sukop et al. [14], a method for creating a basic model of 2D fluid flow is provided, which maps the possible collisions that can occur and the outcomes that are determined by a set procedure. It is the randomness generated by these procedures that is essential to its ability to simulate flows. This procedure does provide reasonable results; however, with the standard of realism ever-increasing, its ability to provide a realistic model is substantially limited.

A similar model was made to represent the effect of polymer chains on fluid flow [15] where a lattice-gas CA was used to provide a 2-dimensional model. It was noted that further work must be done to develop a method of using the lattice-gas method to provide a 3-dimensional model that was able to provide realistic results with a reasonable computational effort.

In a paper by Koelman and Nepveu [16] they demonstrated how it is possible to use a CA to model flow through a porous material. They were able to model a one-phase Darcy automaton based on a Navier–Stokes automaton; however, when they implemented a two-phase Darcy automaton they had to implement much simpler local transition rules. In research presented by Stam [17], the Navier–Stokes equations are used to model the fluid dynamics. While the algorithms implemented do not meet the formalism of CA, they do share several key characteristics. A cell lattice is spanned over the simulation window with each cell holding unique information regarding that particular area. The first difference is that each cell space stores a density value and the horizontal and vertical components of velocity (as well as the z component for a 3-dimensional model). The cell spaces are updated simultaneously at discrete time intervals. In a true CA, each cell can be updated independent of other cells, and the algorithms must solve multiple steps for all cells before the final value is obtained. Nevertheless, Stam's [17] algorithm provided very realistic results with limited computational effort by utilizing a rather basic set of rules, and has potential to be adapted to Cell-DEVS.

In this paper, we use the algorithms presented by Stam to create a CFD solver developed according to the conventions of the Cell-DEVS formalism. These particular algorithms were chosen for several reasons. First, the inherent mathematical stability of these algorithms allows simulations to be advanced using arbitrary time steps. This feature is particularly relevant to the time advancement strategies facilitated by the DEVS formalism. Second, the relative simplicity of these algorithms lends itself well to the prospect of extending this solver to handle increasingly complex scenarios. Third, these algorithms can be performed using a standard PC for reasonably sized grids of both two- and three-dimensions. Fourth, the complete C-code implementation of these algorithms is published in [17], enabling verification of the Cell-DEVS implementation.

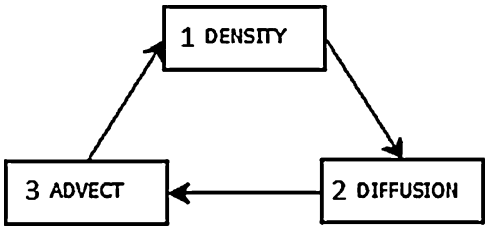

Fig. 1. Density solver steps.

3. Model definition

As previously stated the model will be an adaptation of a model created by Stam, which was based on the Navier–Stokes equations for solving simple fluid flow. The Navier–Stokes equations, named after Claude-Louis Navier and George Gabriel Stokes, make use of Newton's Second law by applying it to fluid flow, and assuming that the stress on the fluid is proportional to the diffusing viscous term and the pressure term. Eq. (1) is the Navier–Stokes equations for velocity and density moving through a velocity field.

$$\frac{\Delta u}{\Delta t} = -(u.\nabla)u + v\nabla^2 u + f$$
$$\frac{\Delta p}{\Delta t} = -(u.\nabla)p + k\nabla^2 p + s \quad (1)$$

The first equation is for solving the velocity vectors; the sum of which is hereafter referred to as the velocity field. The equation is a re-arrangement of the incompressible flow of Newtonian fluids. The acceleration ($\Delta u/\Delta t$) is equal to the sum of; the negative continuity equation ($(u.\nabla)u$, responsible for the conservation of mass), the viscosity ($v\nabla^2 u$) and any body forces present (f). In other words, the change in the evolution of the velocity field is based on the viscosity and any other forces that may act upon it (such as a heating vent). The model treats the fluid space as a 2D grid space.

While this is the most important part of any good CFD solver, it provides very little visually. To make it more useful, we must demonstrate particles moving through the velocity fields. To move objects, we must simply determine what forces are going to be acting on it, and in what direction. These forces are extracted from the velocity fields. Most of the objects we wish to move are relatively light, and the only relevant forces are those applied by the velocity field, such as dust or smoke [17]. One could simply apply these forces to the particles, and see how they move. However, for more complex models, it would be taxing to perform these calculations for a large number of particles. Instead, we could treat the matter as a density of particles, where instead of either being 0 or 1 (no particle, particle respectively); we treat it as a gradient value that ranges from no particles present to some maximum number of particles present. The forces on these densities are applied using Eq. (2), which is similar to the equation used for evolving velocities, but more simplified since the only forces present are solely generated from the velocity vector field.

The fluid is projected as a movement of densities instead of particles, and therefore each cell contains the density value for the given cell area. In Cell-DEVS, each cell must contain all the additional information as well as the set of rules that are used to determine the cell values in the future. The model solves the density in a 3-step process as seen in Fig. 1. The diffusion of the densities is first calculated using Eq. (1). Then the densities are "moved" by examining the forces from the vector field and determining their new locations.

To do this correctly and realistically, the "forces" or the velocity fields must also evolve. The model must create realistic eddies and swirls in the appropriate places. The process of implementing this is even more complicated.

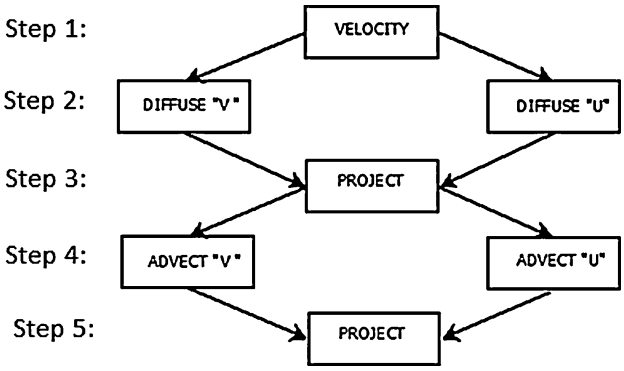

Fig. 2. Velocity solver steps.

The changes in the velocity vectors are caused by three main mechanisms: (i) the addition of forces over time; (ii) the diffusion of the forces; and (iii) the self-propelling nature of the forces. As seen in Fig. 2, the velocity solver has similar steps as the density solver; the diffusion of the forces is calculated similarly to the densities, as well as the advection/movement stage. The new stage is the called the *projection*. As seen in Fig. 2, the projection stage allows for the velocities to be mass conserving. Additionally, this step improves the realism of the model by creating eddies that provide realistic swirling flows.

The diffusion function is responsible for calculating the natural flow of the particles regardless of the forces exerted by the velocity fields. The density for the cell is calculated as the sum of the densities not exiting the cell to the surrounding area and the densities entering the cell from its neighboring cells, as seen in Eq. (2). Eq. (2) states that the new density in the cell at position (i, j) is equal to what will remain in the cell from the original density plus what will enter the cell from the four cardinal neighbors (North, South, East and West). The amount of density that moves between the cells, or viscosity, is determined by the variable a. To increase the resolution of the model this step is run multiple times [17].

$$x(i,j) = \frac{[x(i,j)' + a * (x(i-1,j) + x(i+1,j) + x(i,j+1) + x(i,j-1))]}{1+4a} \quad (2)$$

(Diffusion calculation [17])

The advection function's role is to apply the forces generated by the velocity fields. The force acting on the density at any location is equal to the equivalent velocity vector of u and v. To apply the forces is significantly more complicated. The simplest approach would be to determine the destination based on the magnitude of the forces applied. However, since the system is treated as a cell space and not all densities will end up in the exact center of the cell after moving, this would cause problems. Instead, to move the density, one simply traces backwards from the cell center to compute where the density would have to come from, as seen in Fig. 3 [17].

Then, we take a weighted average of the four cells the densities will be arriving from to calculate the new density at the destination. Once this is done, the cell states are updated with their new densities and the process repeated.

During the calculations of the previous steps, it is rare that the results conserve mass, an important characteristic to maintain realism and stability in the model. The projection function helps conserving mass, and it add some desired visual effects (swirls and eddies). In order for this to occur, the velocity field is defined as the sum of a mass conserving field and a gradient field. To get the mass conserving field, the gradient field is subtracted from the current velocity field. The gradient field is calculated using a linear Poisson system. The projection step is called twice to help maintain accuracy after the advection step.

While the current framework and execution of the CFD model may vary from its predecessor, the results remain consistent. At the end of every cycle, the densities have been diffused and moved,

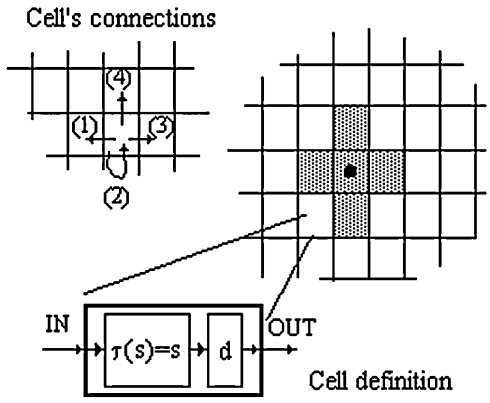

Fig. 3. Tracing backwards to the density source.

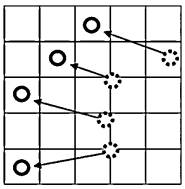

Fig. 4. Cell-DEVS model.

and the velocity fields updated. The external forces and densities are added and ready to begin anew for the next frame.

4. Implementation of rules in cell-DEVS

A Cell-DEVS model is defined as a lattice of cells holding state variables and a computing apparatus, which is in charge of updating the cell states according to a local rule. This is done using the current cell state and those of a finite set of nearby cells (called its neighborhood), as done in Cellular Automata. Cell-DEVS improves CA by using a discrete-event approach which allows better composition. It also enhances the cell's timing definition by making it more expressive. Each cell is defined as a DEVS atomic model, and it can be later integrated into a coupled model representing the cell space. Cell-DEVS models are informally defined as shown in Fig. 4.

The CD++ software [5,6] provides a development environment to create and navigate through the process of Modeling and Simulation (M&S) of a Cell-DEVS model. CD++ is an open-source framework that implements directly the formal specifications of Cell-DEVS, and it has been used to model environmental, biological, physical and chemical models as well as many other real-life simulations. The toolkit includes a high-level scripting language keyed to Cell-DEVS, a simulation engine, a testing interface, and basic graphical interfaces for 2D [6] and 3D [7] simulations.

Models built in CD++ follow the formal specifications of Cell-DEVS and use a built-in language that describes the behavior of each cell [5]. The model specification includes the definition of the size and dimension of the cell space, as well as the shape of the neighborhood and its borders. The cell's local computing function is defined using a set of rules with the form POSTCONDITION DELAY {PRECONDITION}. These rules indicate that when the PRECONDITION is satisfied, the state of the cell will change to the designated POSTCONDITION, whose computed values will be transmitted after consuming the DELAY. If the precondition is false, the next rule in the list is evaluated until a rule is satisfied or there are no more rules [5].

Since the cell states are calculated asynchronously, each cell must contain the following information: (a) the density of the fluid for that cell space; (b) the velocity vectors u and v; and (c) all of the intermediate calculations. The cell space is layered with each layer holding a different piece of information for its corresponding cell as shown in Table 1.

Table 1
Cell space layers and range of values.

Name	Function	Values used
Tempu	Handles the advection of "u" component of the velocity vector	Range: (−2, 2) Positive = Left Negative = Right
Tempv	Handles the advection of "v" component of the velocity vector	Range: (−2, 2) Positive = Up Negative = Down
U	Handles the final projection step and stores the final values for the "u" component of the velocity vectors to be used in the final solution	Range: (−2, 2) Positive = Left Negative = Right
V	Handles the final projection step and stores the final values for the "v" components of the velocity vectors to be used in the final solution	Range: (−2, 2) Positive = Up Negative = Down
Div	Handles the initial projection step defined as *div* in the algorithms	N/A
P	Handles the second projection step, defined as p in the algorithms	N/A
Diff	Handles the diffusion of the densities	Range: (0, 1) values in gradient, 1 is solid particle & <1 is a density
Source	Stores the densities during the density calculations	Range: (0, 1) values in gradient, 1 is solid particle & <1 is a density
Final	Handles the advection of the densities and represents the solution to the density solver	Range: (0, 1) values in gradient, 1 is solid particle & <1 is a density

We have defined the models introduced in Section 3 as a Cell-DEVS model including multiple layers defining different aspects of the model. Each layer focuses on a different aspect of the model, which allows better analysis and visualization. Each of these aspects is based on the information just described on Table 1.

4.1. Diffusion

The diffusion can be calculated by adding the initial density value of the cell to the scaled sum of the densities that could enter that cell from the surrounding cells and then calculating the average. The result is a flow of density from higher to lower concentration. By looping this function, we are able to extend the diffusion to cells outside of the neighborhood; however, with a low value for a cell, these values are negligible at $n \pm 2$ from the cell. The diffusion step is defined as in Section 3.

The implementation of this step is relatively easy. The values of x are stored in the Source layer and the values of x' are stored in the Final layer. For each step the function is run for 20 cycles; on the 20th cycle the value is stored and the cycle is reset. The "*if*" statement in CD++ operates as expected; however, by looking at the timing information we were able to change its behavior to that of a loop, where $n = 20$ and after each cycle it restarts at zero. The resulting code resembles the following:

```
rule:{
    if(remainder(time,20)=0,(0,0,2),
        ((((0,0,2))
        +0.1*(((1,0,1))+((0,1,1))
            +((0,1,1))+((1,0,1)))))/1.4)
} 1 { t }
```

As can be seen, the *time* variable is reset each time that it reaches a multiple of 20 (i.e., 20 cycles passed; we use the *remainder*

function). Otherwise, the new density is recalculated based on the current density and the weighted average of the surrounding cells. In this case, the variable 'a' in the Eq. (2) in Section 3 sets the amount by which the average is weighted, which in this situation is 0.1.

4.2. Advection

The advection step is responsible for the movement of densities and velocity fields. The most obvious method of determining where a *density* will end up is to trace it forward based on the velocity field. However, the method described by Stam [17] suggests that one start in the center of the cell space and trace backwards to find the origins, based on the velocity field. Then, the weighted average of the four closest cells is calculated to determine the source density. This is done because the source is unlikely to be located directly in the middle of the cell, and therefore the surrounding densities will affect the new density values. The advection step, as it appears in the original algorithm in [17] is as follows:

```
void advect ( int N, int b, float * d,
              float * d0, float * u, float * v,
              float dt )                        {
    int i, j, i0, j0, i1, j1;
    float x, y, s0, t0, s1, t1, dt0;
    dt0 = dt*N;
    for ( i=1 ; i<=N ; i++ ) {
        for ( j=1 ; j<=N ; j++ ) {
            x = i-dt0*u[IX(i,j)];
            y = j-dt0*v[IX(i,j)];
            if (x<0.5) x=0.5;
            if (x>N+0.5) x=N+0.5;
            i0 = (int)x; i1=i0+1;
            if (y<0.5) y=0.5;
            if (y>N+0.5) y=N+ 0.5;
            j0 = (int)y; j1=j0+1;
            s1 = x-i0; s0 = 1-s1;
            t1 = y-j0; t0 = 1-t1;
            d[IX(i,j)] =
                s0*(t0*d0[IX(i0,j0)]
                   +t1*d0[IX(i0,j1)])
                   +s1*(t0*d0[IX(i1,j0)]
                   +t1*d0[IX(i1,j1)]);
        }
    }
    set_bnd ( N, b, d );
}
```

In order to model the advection step in Cell-DEVS, we are required to include any of the cells in the neighborhood where the density can originate. The first and most important part is ensuring that the possible source cells are included within the neighborhood. The neighborhood of the advection step is defined by a 5×5 grid of cells; therefore, the maximum distance that a particle can travel is 2 cells from the center. Hence, the velocity vectors cannot exceed the range of $(-2, 2)$. This can be done by scaling the time step to ensure that the velocities remain within the acceptable limits. For example, if the velocity is 4, it can be scaled down to 2 and the new time step would be half of the original. The values of the cells are truncated to discrete values so that there are 5 potential values (-2, -1, 0, 1 and 2) for u and v, and thus 25 different combinations of truncated u and v values. The ratio of the four source cells is calculated for each combination. The following is a portion of the code that determines the value if the truncated values of the u and v velocities are both 1.

The following code snippet checks to see whether the u and v vectors fall within the range of 1.0–1.999. If this is the case, then the weighted averages are calculated and summed. The complete code contains 25 iterations of the above code segment to cover all possible outcomes. This function is used 3 times in each cycle; the advection of the density, the advection of u and the advection of v. Since the offsets of the required planes are the same for both u and v (the offset is 0), the function can be recycled to solve for both. The advection of the density step requires access to a different plane with a different offset (2). We store two values on each cell; we access to the values with the *trunc* and *remainder* functions.

```
if(trunc((0,0,-6)) = 1,
    if( trunc((0,0,-5)) = 1,
        (((1 - remainder( abs((0,0,-6)),1))
        *((1 - remainder( abs((0,0,-5)),1))
        *(-1,-1,-2)
        +remainder(abs((0,0,-5)),1)*(-1,0,-2))
        +remainder(abs((0,0,-6)),1)
        *((1-remainder(abs((0,0,-5)),1))
        *(0,-1,-2)
        +remainder(abs((0,0,5)),1)*(0,0,-2))
    )
)
```

*Note that the above is just 1 of 24 possibilities

4.3. Projection

The projection step can be broken into three subsections: solving for *div*, *p*, *u*, and *v*. The original algorithm is implemented using the following code from [17]:

```
void project ( int N, float * u, float * v,
               float * p, float * div )
{
    int i, j, k;
    float h;
    h = 1.0/N;
    for ( i=1 ; i<=N ; i++ ) {
        for ( j=1 ; j<=N ; j++ ) {
            div[IX(i,j)] = -0.5*h*(u[IX(i+1,j)]
                                  -u[IX(i-1,j)]+
            v[IX(i,j+1)]-v[IX(i,j-1)]);
            p[IX(i,j)] = 0;
        }
    }
    set_bnd ( N, 0, div ); set_bnd ( N, 0, p );
    for ( k=0 ; k<20 ; k++ ) {
        for ( i=1 ; i<=N ; i++ ) {
            for ( j=1 ; j<=N ; j++ ) {
                p[IX(i,j)] = (div[IX(i,j)]
                              +p[IX(i-1,j)]
                              +p[IX(i+1,j)]
                              +p[IX(i,j-1)]
                              +p[IX(i,j+1)])/4;
            }
        }
       set_bnd ( N, 0, p );
    }
    for ( i=1 ; i<=N ; i++ ) {
        for ( j=1 ; j<=N ; j++ ) {
            u[IX(i,j)] -= 0.5*(p[IX(i+1,j)]
                              -p[IX(i-1,j)])/h;
            v[IX(i,j)] -= 0.5*(p[IX(i,j+1)]
                              -p[IX(i,j-1)])/h;
        }
    }
    set_bnd ( N, 1, u ); set_bnd ( N, 2, v );
}
```

The implementation of *Div* is straightforward. It takes the two u and two v values from their respective *temp* layers and is implemented with the following code in the CD++ Model file:

```
rule:{
    if(remainder(time,20)=0,
        -0.05*((1,0,-4)-(-1,0,-4)
              +(0,1,-3)-(0,-1,-3)),
        (0,0,0))
} 1 { t }
```

As can be seen, the "*if*" function works as a loop that is reset after each iteration. The calculations are essentially the same, the only difference being where and how the information is accessed. The *−4* at the end of each neighbor cell means that those values are taken from the temporary layer for the u vectors while the cells with -3 are taken from the temporary v vectors.

To solve for p, we use the same method as solving for the diffusion. The code segment is exactly the same as mentioned before, however the values of a are adjusted to reflect the viscosity instead of the diffusion coefficient.

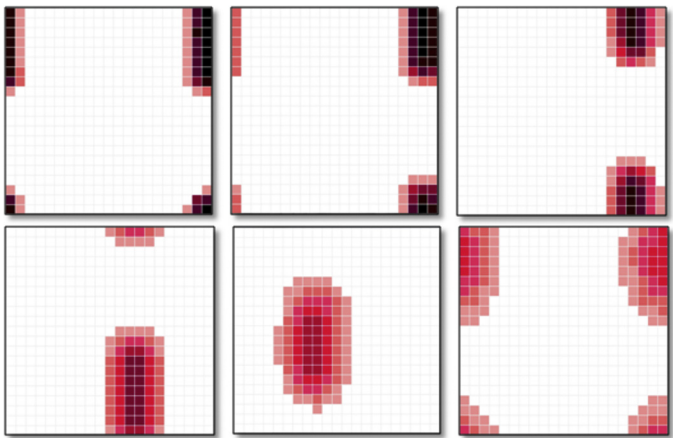

Fig. 5. Progression of diffusion over time from left to right, top to bottom with a coefficient of $a = 0.1$.

The final step for the projection is to separate the vector fields into component form. The horizontal and vertical components are separated as in the algorithm:

```
[u]
rule:{
    if(remainder(time,20)=0,
        if(time=0,(0,0,-2),
           (0,0,-2)-0.05*((-1,0,-6)-(1,0,-6))),
           (0,0,0))
} 0 { t }
[v]
rule:{
    if(remainder(time,20) = 0,
        if(time=0,(0,0,-2),
           (0,0,-2)-0.05*((0,-1,-7)-(0,1,-7))),
           (0,0,0))
} 0 { t }
```

During the projection stage, we had added the velocity vectors together to make a single velocity field. However, for the rest of the algorithm we like to have the velocities in separate fields. These steps will be used twice for each cycle of the model.

5. Simulation results

To test the model, we executed simulations of several scenarios involving different velocities, densities and viscous properties. The simulation results seen in Fig. 5 were initialized as single foci of density with the velocity vectors being randomly generated to have an approximate value of 1 (i.e., a velocity in the upwards diagonal to the left). The diffusion coefficient and the viscosity coefficient were both set to a low value in the range of (0.1). The expected result is that the density foci will spread into more of a cloud, the highest densities being on the leading edge of the cloud as it proceeds to the top left corner.

Fig. 5 shows the results of this particular simulation scenario using a cell space of 21 by 21 cells. The coefficient of diffusion (a) was set to be 0.1. The density field was exposed to a velocity field whose u and v values were randomly set to a range of 0.9–1. The viscosity was set as 1. The results illustrated in Fig. 5 are what we would expect to see in a real situation. The density cloud traveled up and left at an angle of approximately 45° which corresponds to the field applied to it. Additionally, the limited dispersion of the cloud reflects the low diffusive coefficient used. The slight teardrop shape that the cloud took which occurs when the density cloud is moving can be noticed. The concentration was slightly higher on the leading edge and tapered out at the lagging edge, as seen in Fig. 5.

Fig. 6 shows a different simulation scenario, in which we demonstrate how the velocity fields become more regular over time. Since

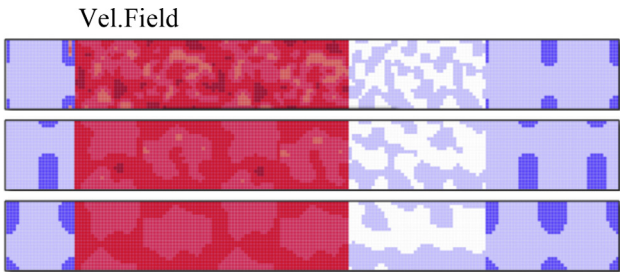

Fig. 6. Evolution of the velocity field and other variables over time from top to bottom (from light to dark represent least to greatest values).

the initial value assignment was random the field would not be stable. As the values did not vary too much (max < 10%), the field soon become more evenly distributed and settled between the range of 0.94–0.96 for the u and v vectors. This is expected since it was a uniform distribution between 0.9 and 1 and with a relatively larger viscosity, the fields would settle quickly.

6. Applications and future work

The ability to solve CFD using the DEVS formalism is particularly relevant to the growing number of researchers in the health and biomedical fields who are now implementing modeling and simulation to enhance their study of human physiological systems. Fluids comprise approximately 60% of human body weight; however, we also move about in fluid surroundings. Consequently, it is no surprise that the inclusion of fluid behavior in digital human modeling is proving vital to properly understand the impact of various disease states on physiological and biomechanical function. For example, CFD modeling and simulation has facilitated the in silico study of blood [18] and air flows [19] within the human, the design and testing of cardiovascular [20] and orthopedic interventions [21], the planning of surgical procedures [22], as well as the impact of shock waves from a blast [23].

Throughout the remainder of this section, we leverage specific examples to illustrate the potential utility of DEVS-based CFD solutions to study mechanics and function in humans. This includes the description of some key work that is underway to enhance the functionality of Cell-DEVS for more advanced human modeling.

6.1. Addressing complexity

A key feature of developing models using the DEVS formalism is the ability to seamlessly couple individual models to build increasingly complex and complete systems. This is a particularly attractive and practical attribute for individuals interested in digital human modeling, since the human body is composed of many complex and interacting systems. This type of modularity lends itself extremely well to composing large-scale models comprised of a number of constituent elements. For example, a module built to study blood flow through a series of vessels could be extended by connecting modules that represent specific organs. Similarly, a model of the lung could be attached to the trachea, throat and nasal cavities to build a more complete model of respiration. In this way, the output of one model can be used as input to another model (or models) as a way of initiating a cascade of events that together comprise a particular phenomenon or behavior, potentially progressing through multiple levels of granularity.

Neighbor ports allow for further compression of the original code so that computations are more light-weight and efficient, as well as easier for developers to understand, when used for models of increased complexity. They are initiated and behave similarly to state variables, but have a few keys differences. The ports will

allow the cells to interact with all the "state variables" of its neighboring cells and perform calculations based on their state values. When a large amount of information must be stored for a cell, we can now create multiple variables/ports to store this information, rather than needing to store the information on multiple planes.

To make use of neighbor ports and state variables there are several syntax changes that must be made to the coding method. A state variable is initialized using the following code:

```
StateVariables: value diffusion velocity
StateValues: -1 3.04 5
```

Similar to regular initial values in the original model, state variables can be loaded from a .var file.

```
InitialVariablesValue: initial_value.var
(0,0) = 1 0 -5
(2,5) = -1 2 4
```

The state variables can be accessed in the local computing function and are implemented by inserting the **$** symbol followed by the desired variable, for example:

```
(0,0) + $diffusion
```

To make changes to state variables a new method was added. The original rules followed the format of:

```
<value> <delay> <condition>
```

The new format is as follows:

```
<value> <assignations> <delay> <condition>
```

Any state changes can be assigned to the state variables. The notation is a semicolon and equal sign,:=, as seen in the following:

```
{(0,0) + $diffusion } { $diffusion :=  $value +
$velocity } 10 { (0,0) > (0,-1) }
```

With state variables only the cell state is recorded in the log file and variables can only be used in the calculations for their own cell state. To avoid this dilemma we have what are called Neighbor Ports. They are initiated similarly to state variables:

```
NeighborPorts: value diffusion velocity
```

To make use of a desired port we simply use the "~" followed by the desired port name, ~diffusion. The syntax for the rules is once again modified to look as follows:

```
<port_assignations> [ <assignations> ] <delay>
<condition>
```

For example:

```
{ ~value := ~diffusion + ~velocity , ~diffusion
:= 4, ~velocity := (0,-1)~diffusion } 10 {
(0,1)~value > 5 }
```

A benefit to the original model was that it provided an in-depth view at the inner workings of the model, allowing us to see how the velocity fields changed over time and not just looking at the end result. However, one drawback was that a large amount of space was required to store this plethora of information. When the model is implemented with neighbor ports, the majority of the same information is still stored; however, it is now stored in a single cell, instead of multiple cells on multiple planes. Since a large amount of information is being produced for each cell a new method is used. Every cell now has a dedicated log file, which allows for more overall information to be stored. As previously mentioned, it may be beneficial to see the behavior of other variables during the simulation.

6.2. Introducing barriers

Fluid-structure interaction plays a key role in numerous physiological phenomena throughout the human body, making the ability to model structural barriers extremely important. Anatomical boundaries compose the pathways through which air, blood, and nutrients flow. To complicate things, no single boundary type exists in the human body. Tissue barriers can possess different material properties (rigid or elastic) and can even transition from rigid to soft (or vice versa). To illustrate, the hard palate (bony front portion of the roof of the mouth) transitions to become the soft palate toward the rear of the mouth. The soft palate is movable by the underlying muscles, which enables it to fulfill its role of closing the nasal passages during swallowing. As with any other barrier comprised of, or containing, muscular elements (e.g., heart, blood vessels), the stiffness of that boundary will vary according to the level of muscle activity. This will, in turn, impact how solids and fluids interact with the tissue barriers. In certain cases, a lack of muscle tone can cause that particular structure to collapse, which can dramatically alter fluid mechanics [24]. All of these factors make for a very complex set of fluid–structure interaction problems that occur within the human.

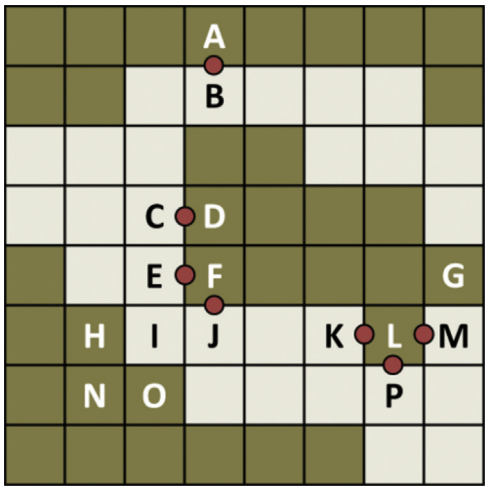

Fig. 7. Model with solid (dark) and fluid (light) cells. (For interpretation of the references to color in text, the reader is referred to the web version of the article.)

One way to add barriers to the Cell-DEVS CFD model is to classify every cell as a fluid cell or a solid cell. In Fig. 7, dark cells such as **A** and **D** are solid cells whereas light cells such as **B** and **C** are fluid cells. All cells still have u, v, p, and *div* values that describe flow. The difference is that for fluid cells, these values represent the real-world movement of fluid. For solid cells, these values have no real-world meaning, but they provide a simple and efficient way to influence the flow in the fluid cells in a manner that produces the effect of barriers.

The popular way to model the effect of a barrier is known as the *no-slip condition*. The no-slip condition has a long history, as described in [25], but the basic idea is that the fluid velocity is always zero at a fluid–solid interface. This model explains why dust can remain on the surface of a fan's blades even when the fan is spinning. To understand how this could be implemented in a Cell-DEVS model, consider the interface between cells **A** and **B** in Fig. 7. The no-slip condition implies that if the velocity were to be calculated at the center of the interface, indicated by a red dot, that velocity should be zero in both the horizontal and vertical directions. The simplest way to approximate the velocity at that point is to average the known velocity at the center of fluid cell **B** with the virtual velocity at the center of solid cell **A**. The average is not actually computed; rather, the velocity at cell **A** is overwritten with the negation of the velocity at cell **B**. Thus, if the final cell velocities were to be averaged, the resulting velocity pertaining to the interface would be zero.

With the technique described above, it is very simple to influence the flow in cell **B** by modifying the velocity of cell **A**, or to influence the flow in cell **C** by modifying the velocity of cell **D**. The

flow around solid cell **F** is more complicated because there are two adjacent fluid cells. The velocity of cell **F** should be the average of the negated velocities of cells **E** and **J**. A discrepancy between these velocities will introduce error, and as a result the no-slip condition will not be strictly met at the interfaces indicated by the two red dots. The error can be reduced by increasing the resolution of the model.

Cell **L** is similar to cell **F**, but more serious in terms of accuracy due to the fact that three negated velocities must be averaged: those of cells **K**, **P**, and **M**. An alternative to performing this three-way average is to simply restrict cell-space models such that every solid cell has at most two neighboring fluid cells. One could go a step further and disallow cases such as cell **G** in which a solid cell has fluid cells on opposite sides. The basic rule is that solid obstacles should be at least two cells thick at every point and along either axis. Arguably, fluid passages should also be several cells thick, unlike the example where flow may be unrealistically dampened at cells **B** and **I**. For all realistic models, increasing the resolution will help satisfy these constraints.

Another complication pertains to solid cells similar to cell **N** that are only diagonally adjacent to fluid cells. The velocities assigned to these cells should be average of the velocities assigned to a subset of the adjacent solid cells. The subset includes only cells that are themselves adjacent to fluid cells. For example, one would first compute the velocities of cells **H** and **O**, then average them to obtain the velocity of cell **N**.

The rules described above address the no-slip condition, but an additional step is required to enforce conservation of mass and prevent flow into and out of solid regions. While the no-slip condition will have the effect of slowing fluid in the vicinity of barriers, conservation of mass will tend to promote flow in a tangential direction to the barriers. For example, it will guide flow through passages. To implement the step, one replaces the p and *div* variables of solid cells by averaging the values of the adjacent fluid cells. Note that these values are not negated, but otherwise the procedure is completely analogous to that adopted for the velocity vectors. Essentially, this step eliminates any pressure difference across a solid–fluid interface.

Barriers are implemented with a suitable definition of the `set_bnd` function, invoked throughout the original CFD code in Section 4. For the sake of brevity, the definition of `set_bnd` published in [17] assumed a one-cell border of solid cells on the outside of the cell-space. To represent internal boundaries, one must redefine the function to cover all cases described above. The convention in the paper was that a second parameter value of 1 was used to propagate the negation of the u component of fluid velocities, a value of 2 was used to propagate the negation of the v component, and a value of 0 was used to propagate p and *div* values without negation. Suitable conventions will have to be developed for the corresponding operations in the Cell-DEVS model.

6.3. Enhancing time advancement

Human physiological systems often interact with one another to create cascades of events, or a set of necessary conditions for a particular phenomenon to occur in an ideal fashion. For example, the pharynx (part of the throat) provides a conduit for two distinct functions that must be separated: (i) the passage of air, during breathing; and (ii) the transport of boluses of food and liquids during swallowing. Normally, swallowing occurs following exhalation, a valve (i.e., the epiglottis) blocking the airway to the lungs so that a bolus can pass safely through the throat and into the digestive tract without any food or liquid passing into the lungs. In this sense, exhalation can help trigger a closing of the valve that facilitates normal swallowing. Thus, event-based initiation of other events serves as an important component of coordinated function.

In some instances, the speed at which a specific phenomenon or event occurs can vary according to a particular set of state conditions. Differential flow rates occur relative to distance from the barriers (i.e., slower flow closer to a barrier than toward the center of a tube), the size of the structures through which a fluid flows (e.g., radius of a tube), as well as according to particular functional activity. To illustrate the latter, the speed at which air is exhaled will be relatively slow during resting state breathing, but will increase during speech, and reaches extreme speeds during a cough or sneeze. Based on these differences in speed, air flow solutions would not need to be computed at the same time resolution, but could be modified according to the scenario in order to control the computational load.

In the CFD algorithm, every cell has a corresponding velocity. Multiplying this velocity by the time step gives a displacement vector. The basic tradeoff is as follows: a large time step will produce large displacements, which increases efficiency but compromises accuracy; a small time step will produce small displacements, which decreases efficiency but improves accuracy. A naïve CFD algorithm would also have the drawback of becoming unstable for large displacements. The velocities could start dramatically increasing in a self-exacerbating cycle, eventually causing numerical overflow. This will not occur with the algorithm of [17] because it is intrinsically stable, but large time steps can still lead to inaccurate results.

The use of DEVS is advantageous in that the formalism inherently accommodates variable time steps. It is very practical to impose an upper limit on the magnitude of the displacement vectors. If the fluid is moving sufficiently slowly at every cell, time advances in fixed steps according to a simulation parameter. But if the flow rate increases such that the displacement at any cell exceeds the displacement limit, the time step is reduced such that the limit is not crossed. This time advancement pattern refines temporal resolution and improves accuracy during periods of rapid motion, such as a sneeze. It then coarsens temporal resolution and increases computational efficiency during periods of relative stagnation, such as resting state breathing.

The use of Cell-DEVS produces an additional incentive for adapting the time step. By limiting the magnitude of the displacement vectors, one can define a smaller cell neighborhood. This leads to simpler code, as well as faster execution of individual time steps. If the displacement limit is equal to the width of one cell, minus some small epsilon, a 3×3 cell neighborhood is adequate for each step: diffusion, advection, and projection.

Like most CFD algorithms, that of [17] is implemented using fixed time steps. However, the diffusion and advection functions take the time step as an argument, making them easy to deploy with an adaptive time strategy. The projection step is independent of the time step.

It is worth noting that even with the use of a variable-time-step, the Cell-DEVS CFD model can still interact with other DEVS models that use different time advancement schemes. It is possible to use a fixed-time-step finite element analysis model to allow the barriers to move in response to fluid pressure and other forces. If the simulation captures signaling from a medical device or the human brain, the signals can be scheduled by another model in classical discrete-event fashion. As mentioned at the outset, the fact that all of these models can be coupled together is the primary rationale for modeling the physics of motion using DEVS-based methodologies and tools.

7. Conclusions

The CFD solver implemented with Cell-DEVS represents a step toward introducing a formal discrete event framework to a large

class of applications for which the physics of moving solids and fluids must be simulated. Transition rules implemented using the CD++ toolkit, and applied uniformly over a cell space, produced the same fluid flow patterns as traditional CFD algorithms that have been implemented using explicit loops and dedicated simulators. Additional work is needed to incorporate barriers that restrict flow, and to explore the use of variable time steps to concentrate computational effort on periods associated with high flow rates. The ability to seamlessly couple dramatically different types of models, from CFD and solid deformation solvers to models of medical devices and other artificial systems, may aid in the simulation of the complex systems encountered in biomechanics and other engineering fields.

Acknowledgments

This work has been partially funded by NSERC and Autodesk Research. The authors want to thank Jos Stam for his valuable support in providing the original CFD source code along with numerous insights into CFD algorithms.

References

[1] B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of Modeling and Simulation, 2nd ed., Academic Press, London, 2000.
[2] H.L.M. Vangheluwe, DEVS as a common denominator for multi-formalism hybrid systems modeling, in: Proceedings of the IEEE International Symposium on Computer-Aided Control System Design, Anchorage, AK, 2000, pp. 129–134.
[3] J.D. Anderson, Basic philosophy of CFD, in: J.F. Wendt (Ed.), Computational Fluid Dynamics, 3rd ed., Springer-Verlag, Berlin/Heidelberg, 2009, pp. 3–14.
[4] A. Ilachinski, Cellular Automata: A Discrete Universe, World Scientific Publishing Co., Singapore, 2001.
[5] G.A. Wainer, Discrete-Event Modeling and Simulation: A Practitioner's Approach, CRC Press, Boca Raton, FL, 2009.
[6] M. Bonaventura, G. Wainer, R. Castro, A graphical modeling and simulation environment for DEVS, Simulation 89 (January (1)) (2013) 4–27.
[7] G. Wainer, Q. Liu, Tools for graphical specification and visualization of DEVS models, Simulation 85 (3) (2009) 131–158.
[8] M. Van Schyndel, G. Wainer, M. Moallemi, Computational fluid dynamic solver based on cellular discrete-event simulation, in: Proceedings of Simultech 2013, Rejkyavik, Iceland, 2013.
[9] S. Chen, G.D. Doolen, Lattice Boltzman Method for Fluid Flows, Annu. Rev. Fluid Mech. 30 (1998) 329–364.
[10] Currie, Fundamental Mechanics of Fluids, McGraw-Hill, New York, 1974.
[11] E.F. Toro, Rienmann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction, 3rd ed., Springer-Verlag, Berlin/Heidelberg, 2009.
[12] J.M. Saleh, Fluid Flow Handbook, McGraw-Hill, New York, 2002.
[13] U. Frisch, B. Hasslacher, Y. Pomeau, Lattice-gas automata for the Navier–Stokes equation, Phys. Rev. Lett. 56 (1986) 1505–1508.
[14] M.C. Sukop, D.T. Thorne Jr., Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers, Springer, Berlin, 2006.
[15] J.M.V.A. Koelman, Cellular-automata-based computer simulations of polymer fluids, Numer. Methods Simul. Multi-Phase Complex Flow Lect. Notes Phys. 398 (1992) 146–153.
[16] J.M.V.A. Koelman, M. Nepveu, Darcy flow in porous media: cellular automata simulations, Numer. Methods Simul. Multi-Phase Complex Flow Lect. Notes Phys. 398 (1992) 136–145.
[17] J. Stam, Real-time fluid dynamics for games, in: Proceedings of the Game Developer Conference, San Jose, CA, 2003.
[18] J.D. Müller, M. Jitsumura, N.H.F. Müller-Kronast, Sensitivity of flow simulations in a cerebral aneurysm, J. Biomech. 45 (2012) 2539–2548.
[19] S.K. Kim, Y. Na, J.I. Kim, S.K. Chung, Patient-specific CFD models of nasal airflow: overview of methods and challenges, J. Biomech. 46 (2013) 299–306.
[20] G. Janiga, C. Rössl, M. Skalej, D. Thévenin, Realistic virtual intracranial stenting and computational fluid dynamics for treatment analysis, J. Biomech. 46 (2013) 7–12.
[21] A. Hölzer, C. Schröder, M. Woiczinski, P. Sadoghi, P.E. Müller, V. Jansson, The transport of wear articles in the prosthetic hip joint: a computational fluid dynamics investigation, J. Biomech. 45 (2012) 602–604.
[22] G. Mylavarapu, M. Mihaescu, L. Fuchs, G. Papatziamos, E. Gutmark, Planning human upper airway surgery using computational fluid dynamics, J. Biomech. 46 (2013) 1979–1986.
[23] J.C. Robert, T.P. Harrigan, E.E. Ward, T.M. Taylor, M.S. Annett, A.C. Merkle, Human head-neck computational model for assessing blast injury, J. Biomech. 45 (2012) 2899–2906.
[24] P. Anderson, S. Fels, S. Green, Implementation and validation of a 1D fluid model for collapsible channels, J. Biomech. Eng. 135 (2013), 111006 (7 pp.).
[25] M.A. Day, The no-slip condition of fluid dynamics, Erkenntnis 33 (1990) 285–296.

Michael Van Schyndel received a B.M.E. and a M.A.Sc. (Biomedical Engineering) at Carleton University under the supervision of Professor Gabriel Wainer. He has worked extensively in the Cell-DEVS field creating many models including building evacuation and predator–prey relationships.

Gabriel A. Wainer is a professor at SCE, Carleton University. He is the author of 3 books and over 270 research papers. He is one of the founders of SIMUTools, SimAUD and the Symposium of Theory of Modeling and Simulation. He is a Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE CISE, and Wireless Networks. He has been the recipient of the IBM Eclipse Innovation Award, Carleton University's Research Achievement Award (2005, 14), the First Bernard P. Zeigler DEVS Award, the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award and the SCS Distinguished Professional Award (2013).

Rhys Goldstein is a principal research scientist with the Ergonomics & Environment Research Group at Autodesk Research in Toronto, Canada. His work combines building information modeling (BIM) with sensor data and simulation to help predict and reduce energy consumption in buildings. He is currently investigating the use of a set of modeling conventions known as the Discrete Event System Specification (DEVS), to help researchers collaborate in the development of simulation software. Rhys received a B.A.Sc. in Engineering Physics at the University of British Columbia in 2003, and a M.A.Sc. in Biomedical Engineering at Carleton University in 2009.

Jeremy P.M. Mogk is a senior research scientist with the Ergonomics & Environment Research Group at Autodesk Research, and a key contributor to the Parametric Human Project. His interests include human musculoskeletal modeling, and the generation and tuning of person-specific biomechanical models of the upper extremity. Currently, he is working to develop an ontology of human musculoskeletal anatomy that describes the geometric variability between individuals. Jeremy was a Postdoctoral Research Associate at the Rehabilitation Institute of Chicago. He holds M.Sc. and Ph.D. degrees in Biomechanics from York University, as well as a Bachelor's degree in Kinesiology (BKin) from McMaster University.

Azam Khan is the head of the Environment & Ergonomics Research Group at Autodesk Research. He is the founder of the Parametric Human Project Consortium, the Symposium on Simulation for Architecture and Urban Design (SimAUD), and the CHI Sustainability Community. He is also a founding member of the International Society for Human Simulation and is currently the Velux Guest Professor at The Royal Danish Academy of Fine Arts, School of Architecture, at the Center for IT and Architecture (CITA) in Copenhagen, Denmark. Azam received his B.Sc. and M.Sc. in Computer Science at the University of Toronto.