Computer Science (Datavetenskap)
20 credits (20 poäng)
October, 2006

The Algorithms and Implementation of `MesoRD`

Johan Hattne

Department of Information Technology, Uppsala University

Supervisor (Handledare):
Johan Elf, Department of Cell and Molecular Biology
Scientific reviewer (Ämnesgranskare):
Per Lötstedt, Department of Information Technology, Scientific Computing

Abstract

The next subvolume method, NSM, is a novel procedure for the simulation of the Markov process corresponding to the reaction–diffusion master equation. Unlike many other stochastic simulation algorithms, the NSM does not require spatial homogeneity, and is therefore applicable to a wider range of models. `MesoRD`, a portable, open source software implementing the NSM, attempts to make the method available to a larger audience. The aims are (i) to permit as general input model descriptions as possible, (ii) interacting with the user through sensible interfaces, while (iii) preventing the two aforementioned goals from negatively impacting performance. This text describes the design of `MesoRD` and its underlying algorithms, and discusses possible directions for future development.

Contents

Chapter 1

Introduction

1.1 Mission Statement

This report describes `MesoRD`, a software tool for stochastic simulation of reactions and diffusion in space. `MesoRD` is an attempt to implement the next subvolume method [1, 2] efficiently, yet user-friendly. The problem under consideration is that of finding the molecular population levels in some structured volume, V, at any time, $t > t_0$, given the number of molecules interacting in V as well as their spatial distribution at $t = t_0$, see Fig. 1.1. No actual calculations will be presented – the sole concerns are the theory and the implementation of the algorithm.

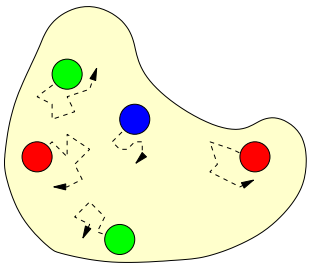


Figure 1.1: The volume V is illustrated by the shaded shape. Inside reside five molecules of three different species. Diffusive movement is indicated by the dashed arrows. If two or more molecules come sufficiently close during simulation, they may react. If this is what a system looks like at $t = t_0$, `MesoRD` attempts to determine what it *could* look like at any later time, $t > t_0$.

To put the aim of the work documented here into theoretical context, one should note that it focuses on one of several fundamentally different views of spatially distributed, coupled systems of chemical reactions.

Macroscopic Here, one regards chemical systems in terms of continuous concentrations of molecular species. The concentrations change in time according to differential equations derived from a set of reactions. Historically, this deterministic approach has been the most commonly adopted view, probably because of its mathematical simplicity.

Mesoscopic An increase in level of theory can be brought about by keeping track of *every single molecule of interest*, changing the composition of the system by means of stochastic processes. These processes are derived from the reactions occurring in the system, just like the differential equations in the deterministic approximation.

Microscopic This very detailed view applies the laws of physics to the system's molecular or atomic constituents. Using quantum theory on electrons, or Newtonian mechanics on individual atoms, yields methods that, while highly accurate, are computationally feasible only for small systems [3]. The systems of interest here are too big to be manageable by microscopic methods.

The approaches approximate each other in order of increasing level of theory; the first, deterministic, approach can be viewed as an approximation of the second [4]. However, the second, stochastic, view cannot be treated as an approximation of the first. Deterministic and stochastic approaches are further reviewed and contrasted by Lok and Brent [5].

As may now be apparent, a balance between level of detail and system size needs to be established. Big systems, rich in detail, would lead to an explosion in complexity, which cannot currently be coped with.

The stochastic procedure is more broadly applicable and is, in many respects, more natural [6]. It has appealing advantages for certain systems, such as small biological cells. In these systems, molecular diffusion can be slow and the number of molecules of a certain species is often very small. Additionally, the rates of chemical reactions typically depend non-linearly on local species composition; tiny fluctuations in the configuration can have tremendous effects on the overall state. Fluctuations here, even though irreproducible, are *neither* undesirable artifacts *nor* necessarily due to imperfections in the observational technique, but inherent properties of any real chemical system, and contribute to the system's characteristics [7, 8, 9].

This chapter briefly presents the algorithms behind the next subvolume method. Their implementations and a number of other techniques used in `MesoRD` to enable stochastic simulation in practise are discussed in the remaining chapters. `MesoRD` represents an ongoing effort – it is not yet ready for production use. Even though the algorithmic base may not change significantly as work progresses, there is ample space for improvement, particularly in the areas of efficiency and user-friendliness.

1.2 The Master Equation

The purpose of `MesoRD` is the simulation of a set of M reaction events, $\{R_\mu\}_{\mu=0}^{M-1}$, involving a set of $\mathcal{S}$ diffusing species, $\{S_i\}_{i=0}^{\mathcal{S}-1}$, in a well-defined volume, V. Such a reaction may be written

$$S_i \xrightarrow{k_1} S_j, \tag{1.1}$$

which means that every time unit, a number of molecules of species S_i are transformed to molecules of species S_j. The number, or equivalently, the *rate* at which the reaction proceeds, is expected to be proportional to k_1. One cannot

hope to know this number exactly, because it depends on many factors, several of which may be unknown. Therefore, the introduction of probabilities should come as no surprise. Now k_1 is reinterpreted as the compound probability of occurrence of a transformation event, which will turn one molecule of S_i into a molecule of S_j.

For reactions involving more than one reactant species, one would suspect that the reactants need to be spatially close in order to react. For example, the rate of the reaction

$$S_i + S_j \xrightarrow{k_2} S_k \tag{1.2}$$

is expected to be proportional to k_2. If the concentrations of S_i, S_j and S_k in Eq. 1.2 are denoted by s_i, s_j and s_k respectively, one would expect that for some function f

$$\text{rate} = \frac{\mathrm{d}s_k}{\mathrm{d}t} = k_2 \cdot f\left(s_i,\ s_j\right),$$

but *only* as long as the reacting S_i and S_j molecules are sufficiently close. The proximity requirement holds for any number of reactants. However, most multi-reactant reactions occur in a sequence of elementary steps, where each step involves only one or two entities [7, 10].

Molecules move relative to each other by means of diffusion. The diffusive movement of molecules in space can also be described using rates, or probabilities. Reaction and diffusion is illustrated in Fig. 1.2.

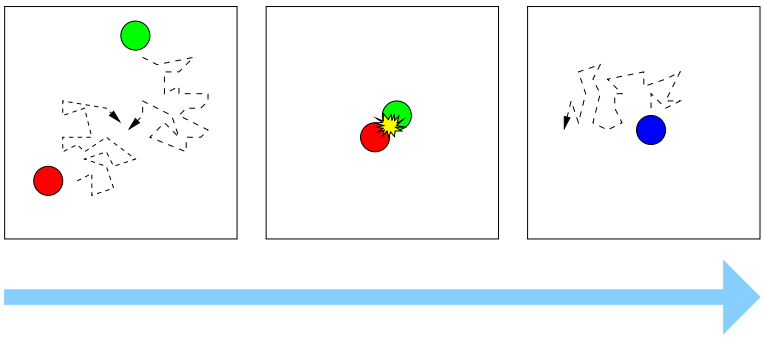


Figure 1.2: Reaction and diffusion in space. The square delimits the system, which is shown at different points on the light blue time axis. Initially, there are two molecules of different species in the system. At first, the red and the green molecules are spatially well separated and therefore unable to react. By diffusive movement, they are brought into proximity. The red and the green molecules react, and transform into one blue molecule, which diffuses through the system.

If one knew the location of every molecule at some time t, one would be in a position to give the *state* of the system. Denote this time-dependent state $\sigma = \sigma\left(t\right)$. Assume that, given an initial state σ_i, it is possible to calculate the probability for a transition to a new, final, state, σ_f, *without* having to consider the states that lead up to σ_i. It can be shown that many chemical systems have this Markovian property [11]. Denote the probability of being in an arbitrary

state σ_k at time t by $P(\sigma_k,\ t)$, and the probability of transition from σ_i to σ_f in some time interval of length $\mathrm{d}t$ by $W(\sigma_i \to \sigma_f)\,\mathrm{d}t$. The Markovian master equation can now be defined as

$$\frac{\partial P(\sigma_f,\ t)}{\partial t} = \int_{\sigma_i} W(\sigma_i \to \sigma_f)\, P(\sigma_i,\ t) - \int_{\sigma_i} W(\sigma_f \to \sigma_i)\, P(\sigma_f,\ t)\,. \qquad (1.3)$$

For all but the very simplest of systems, the Markovian master equation is multivariate [12]. At steady-state, or thermal equilibrium, the left-hand side of Eq. 1.3 is zero. This could be ensured by requiring that detailed balance,

$$W(\sigma_i \to \sigma_f)\, P(\sigma_i,\ t=\infty) = W(\sigma_f \to \sigma_i)\, P(\sigma_f,\ t=\infty)\,,$$

holds at $t = \infty$.

Solving the master equation would give the time evolution of the state probability function, $P(\sigma_k,\ t)$, that is, the time evolution of the location of every particle in the system. To solve the master equation, one would need to compute the transition probability for every transition pair $(\sigma_i,\ \sigma_f)$. This is a daunting task. As shall be seen in section 1.5, the state space of reasonable systems is far too complicated for Eq. 1.3 to be solved analytically. Thus, `MesoRD` simulates, and produces *trajectories* in state space. These trajectories are particular realisations of the time evolution of the system's master equation. The master equation may perhaps be seen as the Schrödinger equation of reaction–diffusion kinetics: only in the most trivial cases can it be solved exactly[1].

In what remains of this text, it will sometimes be necessary to distinguish concentrations from absolute, integer, numbers of molecules. In such cases, $^{\#}s_i$ will denote the copy number – the number of S_i molecules – while s_i shall be interpreted as the corresponding concentration. In general, roman indices are used for species while Greek symbols are used to index reactions. Lowercase bold symbols, such as $\mathbf{u}$ and $\mathbf{v}$, are reserved for vector quantities, while uppercase bold symbols, for example $\mathbf{M}$ and $\mathbf{X}$, denote matrices. Unless specifically stated otherwise, logarithms are taken base two[2], and indices into arrays, vectors and matrices start at zero, not one.

1.3 Gillespie's Direct Method

The theoretical foundation of what follows was established by the 1960's; McQuarrie's review gives a detailed account of the history of mesoscopic kinetics, and provides many references its origin [7]. Actual simulation algorithms started to appear some years later. The precursor of all simulators described here is a computational procedure developed independently by several groups in the 1970's, mainly Bunker *et al.* [13] and Bortz *et al.* [14]. The method was popularised in 1976, when Gillespie reinvented it and demonstrated its capabilities in real applications [4, 9]. Like most mesoscopic simulation efforts since,

[1] Where, if the analogy is taken a bit further, the reaction and diffusion probabilities take the place of the wave function in the Schrödinger equation [6].

[2] We note in passing that there exist but `10` kinds of people: those that think $\log(x) = \log_2(x)$ and those that do not. The former are probably also more comfortable with indexing from zero.

the Gillespie method starts by requiring spatial homogeneity. This is a serious restriction.

Using assumptions similar to those one would deploy in deriving reaction rates in a hard-sphere molecular model, Gillespie argues that one can express the probability that any particular combination of molecules will react according to a particular reaction in a finite time interval, $\mathrm{d}t$ [4, 10].

Definition 1.3.1 (Gillespie's Fundamental Hypothesis)**.** $c_\mu \mathrm{d}t \equiv$ average probability, to first order in $\mathrm{d}t$, that a particular combination of reactant molecules will react according to reaction R_μ in the next time interval $\mathrm{d}t$.

The probability that a given reaction R_μ will occur depends on the details of the reaction as well as the instantaneous numbers of reactant molecules in the reaction volume. This dependence on reactant concentration is captured by the c_μ factor.

The number of molecules also affects the probability in a combinatorial fashion: the more different possibilities of combining the reactant particles there are, the more probable the reaction is expected to be. For instance, if reaction R_μ involves two reactant molecules of the same species,

$$2S_i \longrightarrow \text{reaction products},$$

there are $h_\mu = {}^{\#}s_i \left({}^{\#}s_i - 1 \right)/2$ unique reactant combinations. For a reaction involving two molecules of different species, S_i and S_j, the number would be $h_\mu = {}^{\#}s_i \, {}^{\#}s_j$.

Using h_μ with Def. 1.3.1, it is possible to express the probability that reaction R_μ occurs in the time interval $\mathrm{d}t$ as $h_\mu c_\mu \mathrm{d}t$. The difference is that while $c_\mu \mathrm{d}t$ considers one *unique* combination of reactant molecules, $h_\mu c_\mu \mathrm{d}t$ expresses the probability for *any* combination of reactants. If one wishes to track the time evolution, $t > t_0$, of a multi-reaction system, where the state at $t = t_0$ is known, all one needs to know are the answers to two questions [9]:

1. *When* will the next reaction occur?
2. *What* kind of reaction will it be?

An expression for $P(\tau,\ \mu)$ – the probability that reaction R_μ occurs in the time interval $(t + \tau,\ t + \tau + \mathrm{d}\tau)$, where t denotes the current simulation time – would answer both questions.

The sought probability can be expressed by first finding $P_0(\tau)$, the probability that *no* reaction occurs in the interval $(t,\ t + \tau)$. The probability $P_0(\tau)$ in turn, can be calculated by dividing the time interval $(t,\ t + \tau)$ into K subintervals of equal length τ/K. The probability for no reaction in any one of these K intervals is

$$P_K(\tau) = 1 - \sum_{\nu=0}^{M-1} h_\nu c_\nu \frac{\tau}{K} + \mathcal{O}\left(\frac{\tau}{K}\right). \tag{1.4}$$

Equation 1.4 holds for all $K > 1$. Thus, one can let the time intervals become infinitesimal by letting $K \to \infty$. This will remove the interval dependence from $P_K(\tau)$. Applying the standard limit formula [15], and multiplying by $h_\mu c_\mu \mathrm{d}\tau$:

$$P(\tau,\ \mu) = h_\mu c_\mu P_0(\tau)\, \mathrm{d}\tau, \tag{1.5}$$

where

$$P_0(\tau) = \lim_{K\to\infty} P_K(\tau) = \exp\left(-\sum_{\nu=0}^{M-1} h_\nu c_\nu \tau\right). \tag{1.6}$$

Equation 1.5 gives the desired probability. By multiplication by $h_\mu c_\mu \mathrm{d}\tau$, one has constructed the joint probability for two events. The components of the joint probability are:

1. No reaction occurs in the interval $(t,\ t+\tau)$. This probability is given by $P_0(\tau)$ as defined by Eq. 1.6.

2. Reaction R_μ occurs during a time interval of length $\mathrm{d}\tau$. This probability is, according to Def. 1.3.1, given by $h_\mu c_\mu \mathrm{d}\tau$.

Note that Eq. 1.6 depends on the parameters, $\{c_\mu\}$, for *all* reactions and the current copy numbers, through $\{h_\mu\}$, of *all* reacting species.

Equations 1.5 and 1.6 imply that at time t, the next reaction will occur in τ time units, and it will be an R_μ reaction. The values of τ and μ are given by sampling Eq. 1.5, the *reaction probability density function*. Gillespie showed that, given two random numbers r_1 and r_2 in the interval $[0,\ 1]$, τ and μ may be drawn from the reaction probability density function by finding τ and μ such that

$$\begin{cases} \tau = \frac{1}{\sum_{\nu=0}^{M-1} a_\nu} \ln\left(\frac{1}{r_1}\right) \\ \sum_{\nu=0}^{\mu-1} a_\nu < r_2 \sum_{\nu=0}^{M-1} a_\nu \leq \sum_{\nu=0}^{\mu} a_\nu \end{cases} \quad \text{where } a_\mu = h_\mu c_\mu. \tag{1.7}$$

The product $a_\mu = h_\mu c_\mu$ is the state-dependant rate of reaction R_μ[3]. Thus, in Gillespie's direct method, the time until the next reaction is sampled from an exponential distribution with a mean time equal to the inverse of the sum of all reaction rates. Which this next reaction will be is chosen randomly from a distribution determined by the reactions' normalised rates.

Gillespie's algorithm that implements the theory from this section is shown in Alg. 1. The state of the system at time t is kept in the vector $\mathbf{x}_t$, organised such that the i:th component is equal to the copy number of species i at time t. The algorithm needs to be supplied with the rates, $\{c_\mu\}_{\mu=0}^{M-1}$, from Def. 1.3.1. The procedure as shown below assumes the presence of a `random()` function, that returns a random number in the interval $[0,\ 1]$. Random number generation is the only "black box" in Gillespie's algorithm.

By carrying out the procedure from $t = 0$ to $t = t_{\text{stop}}$, one particular realisation of the stochastic process is simulated. An important observation is that the method does *not* make use of fixed time steps. Fixed time steps had previously been found to be the source of computational inaccuracies and instabilities [16].

Gillespie also described a variation of Alg. 1, called the *first-reaction method* [4]. Instead of computing a $(\mu,\ \tau)$ pair, the times, $\{\tau\}$, are computed for every reaction in the system. The reaction with the smallest τ, the *first reaction*, will be the reaction corresponding to the μ as computed by Alg. 1.

[3]Gillespie originally used the term "reaction intensity" for the quantity a_μ, in order to distinguish it from the macroscopic "reaction rate", which would be calculated in exactly the same way. Since both names refer to the same quantity – the number of reaction events per unit time in a given state – there is little reason to introduce additional complexity by using different, context-dependant, names.

Algorithm 1: Gillespie's direct method for spatially homogeneous systems. This algorithm simulates the system in the time interval $[0,\ t_{\text{stop}}]$, and uses two random numbers per iteration. The reaction parameters, $\{c_\mu\}_{\mu=0}^{M-1}$, where M is the number of reactions, are taken from Def. 1.3.1. It takes time $\mathcal{O}(M)$ to update the reaction rates and the state vector, $\mathbf{x}_t$. Finding $(\tau,\ \mu)$ is an $\mathcal{O}(M)$ operation as well.

begin
  for $t \leftarrow 0$ **to** t_{stop} **do**
    for $i \leftarrow 0$ **to** $M-1$ **do**
      $a_i(\mathbf{x}_t) \leftarrow$ The rate of reaction i given the current state $\mathbf{x}_t$
    $\mu \leftarrow \lambda$ such that $\sum_{\nu=0}^{\lambda-1} a_\nu(\mathbf{x}_t) <$ `random()` $\sum_{\nu=0}^{M-1} a_\nu(\mathbf{x}_t) \leq \sum_{\nu=0}^{\lambda} a_\nu(\mathbf{x}_t)$
    $\tau \leftarrow \frac{1}{\sum_{\nu=0}^{M-1} a_\nu(\mathbf{x}_n)} \cdot \ln\left(\frac{1}{\texttt{random()}}\right)$
    Update the state vector $\mathbf{x}_t$ to account for reaction R_μ
    $t \leftarrow t + \tau$
end

Unless there are absolutely no dependencies between the M reactions in the system, Gillespie's first-reaction method wastes random numbers for any system with $M \geq 3$. Curiously, the first-reaction method was initially thought of as little more than a way to gain insight into the direct method. Yet, the first-reaction method is important because it is this formulation that enabled many subsequent improvements.

1.3.1 Macroscopic and Microscopic Reaction Constants

The mathematical relationship between macroscopic rate constants, k_μ, such as those given in Eqs. 1.1 and 1.2, and their corresponding microscopic reaction constants, c_μ, is always rather simple [9],

$$c_\mu = k_\mu \Omega^d,$$

where d is the power of the spatial dimension, and Ω is the reaction volume. A macroscopic rate constant is volume-independent; its use does not require explicit reference to the reaction volume. The reason is that macroscopic rate constants are applied to *concentrations* – numbers of molecules per volume – where the size of the reaction volume is effectively "built in". The probabilities, c_μ, are volume-dependent because they are used together with numbers of molecules which do not carry any information about the reaction volume. With microscopic reaction constants one has to scale the rate constant according to the volume in which the reaction is taking place.

The macroscopic, concentration-based, description has its roots in experimentation where it is impossible to deal with single molecules. From a physical perspective, using c_μ is more convenient, because events are described in terms of a molecular model. Theoretically, c_μ rests on firmer ground than k_μ [4].

1.4 The Next Reaction Method

Gillespie's method gained popularity and was over time applied to systems of an originally unanticipated size. In 2000, Gibson and Bruck introduced the *next reaction method*, which performs significantly better. In their paper, they showed their method to be equivalent to Gillespie's original method [11]. In hindsight, Gibson and Bruck's modifications to the first-reaction formulation of Alg. 1 may seem simple, almost obvious. However, the development did take more than twenty years.

Gibson and Bruck collected a few general ideas that had been introduced over the years to improve the initial algorithm. The list below is not complete, but limited to changes which were eventually incorporated into the next subvolume method.

- Switch from relative time to absolute time. Algorithm 1 computes τ as the *relative* time *until* the next reaction. The *absolute* time of a reaction in Gillespie's method is $t + \tau$, where t is the time when τ is computed. This is still very different from fixed time steps.
- Re-use random numbers where legitimate. Gibson and Bruck showed that it is possible to use certain random numbers more than once. Random number generation is an expensive operation, further discussed in section 2.4.1. The improved algorithm only uses a single random number per iteration.
- Only recalculate quantities if necessary. Gibson and Bruck introduced a static dependency graph to avoid unnecessary recalculation of reaction rates. This means they were able use the first-reaction formulation without wasting random numbers. The purpose of the dependency graph is to keep track of which rates need to be updated after each reaction event. If, for instance, the system contains 100 reactions, only five of which depend on a particular species S_i, then only the rates of the latter five need to be updated after executing a reaction which changes nothing but the number of S_i molecules. Because `MesoRD` currently makes no use of dependency graphs, they will only be mentioned briefly here[4].
- Use appropriate data structures to store quantities that need not be re-computed every iteration.

The basis of Gillespie's approach, the reaction probability density function, given in Eq. 1.5, is kept without modifications.

The improved algorithm typically takes time proportional to the *logarithm* of the number of reactions, M, as opposed to the number of reactions itself. As

[4]Dependency graphs are used in, for instance, compiler design to resolve instruction scheduling. Usually these graphs are implemented as directed, acyclic graphs where the nodes are basic block-instructions at some representation level [17]. The dependency graph in a compiler could implement a Boolean-valued function, `conflicts(`I_a`, `I_b`)`, that returns `true` if instruction I_a must precede I_b. Denoting the number of instructions in a block by n, building the graph requires $\mathcal{O}\left(n^2\right)$ operations, since all pairs of instructions in the block are compared. In Gibson and Bruck's method, the dependency graph is similarly constructed in $\mathcal{O}\left(M^2\right)$ operations where M is the number of reactions [11]. Gibson and Bruck implement two reaction-set valued functions, `Affects(`R_μ`)` and `DependsOn(`R_μ`)`, in place of the simpler `conflicts()` function.

can be seen from Alg. 2 the worst case complexity is still $\mathcal{O}(M)$. This, however, would only occur if all rates need to be recomputed, regardless of what reaction is executed.

Algorithm 2: The next reaction method by Gibson and Bruck. The details of the dependency graph are not shown in the algorithm as given here. Suffice it to state that the update of the dependency graph calls `random()` only once, and has a worst case complexity of $\mathcal{O}(M)$ in the number of reactions, M. The remaining symbols are identical to those in Alg. 1.

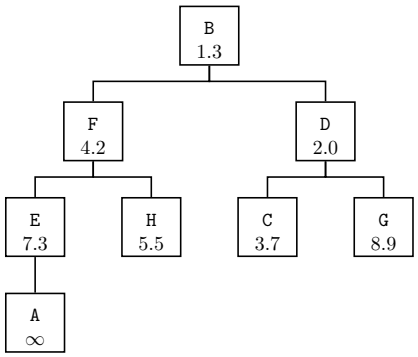
begin
  Generate the dependency graph
  for $\nu \leftarrow 0$ **to** $M-1$ **do**
    $a_\nu(\mathbf{x}_0) \leftarrow$ The rate of reaction ν given state $\mathbf{x}_0$
    $\tau_\nu \leftarrow \frac{1}{a_\nu(\mathbf{x}_0)} \cdot \ln\left(\frac{1}{\texttt{random()}}\right)$
  for $t \leftarrow 0$ **to** t_{stop} **do**
    $(t,\ \mu) \leftarrow (\tau_\nu,\ \nu)$ where $\tau_\nu = \min_{0 \leq \lambda \leq M-1}(\tau_\lambda)$
    Update the state vector $\mathbf{x}_t$ to account for reaction R_μ
    Update $\{\tau_\nu\}$ according to the dependency graph
end

Gibson and Bruck's algorithm introduces an implicit requirement for avoiding re-computation of $\{a_\mu\}$: the $\{c_\mu\}$ must not change. If it were the case that the $\{c_\mu\}$ were varying over the course of the simulation – for example because the reaction volume expands or contracts, or the temperature changes – all the $\{a_\mu\}$ would have to be recomputed, irrespective of what reaction was executed. Thus, what was gained in Alg. 2 compared to Alg. 1 would be lost. In Gillespie's original algorithm, time-dependant $\{c_\mu\}$ would not have impacted performance, because the $\{a_\mu\}$ are updated in every iteration [4].

1.4.1 The Event Queue

Another key improvement in Gibson and Bruck's algorithm is not visible in Alg. 2. It concerns the first and last step in the time loop: how to find τ_μ and μ for which $\tau_\mu \leq \tau_\nu,\ 0 \leq \nu \leq M-1$, and consequently updating the dependencies of R_μ. A naive implementation would store the $(\tau_\nu,\ \nu)$ in a linear array. Finding the smallest τ_ν would require a scan of the entire array – an $\mathcal{O}(M)$ operation. Principles from standard sorting algorithms can be put to use in order to reduce this to $\mathcal{O}(\log(M))$.

The idea is to store the $(\tau_\nu,\ \nu)$ in a *heap*. The (binary) heap data structure is an array that can be viewed as a nearly complete (binary) tree [18]. The tree is completely filled at all levels, except possibly the lowest. For an array of length n, the corresponding binary tree will have height $\log(n)$. The relationship between the nodes of the tree and the elements in the corresponding array, A, is simple: the root of the tree is $A[0]$, because in this text indexing starts at zero. The parent of a node $A[i]$ is $A[\lfloor i/2 \rfloor]$, its left and right children are $A[2i]$ and $A[2i+1]$ respectively. The two interpretations of a heap are illustrated in Fig. 1.3.



(a) Tree structure of a heap.

B	F	D	E	H	C	G	A
1.3	4.2	2.0	7.3	5.5	3.7	8.9	∞

(b) Array structure of the heap in Fig. 1.3(a).

Figure 1.3: A sample heap represented as a binary tree is shown in Fig. 1.3(a). The reactions referred to by the nodes are indicated by letters. The times of the reactions are given by the numbers in the nodes. In an actual implementation, the letters may correspond to pointers in memory to information concerning the reactions. The array structure corresponding to the tree is shown in Fig. 1.3(b).

The purpose of the heap in the next reaction method is to keep track of reactions. Only two operations need to be performed on the heap: finding the reaction with the earliest time and updating the heap after the time of a reaction has changed. If the heap satisfies the *heap property*, or, in this context, the *min-heap property*,

$$A\left[\lfloor i/2 \rfloor\right] \leq A\left[i\right],$$

where the inequality is with respect to the time of the reaction in the node, one can see that the root of the tree will always correspond to the earliest reaction. The remaining nodes are sorted only with respect to nodes above or below it, not necessarily with respect to other nodes at the same height.

When viewed as an array, a heap satisfying the heap property is only partially sorted. Because one will only ever need to find the first element, the partiality is sufficient. The heap may therefore be interpreted as a queue, in which only the first element is processed while all others are temporarily ignored[5]. The implication of the queue, for which extraction of the first element is an $\mathcal{O}\left(1\right)$ operation, is that the next reaction in Gibson and Bruck's method can be found in constant time.

[5]A fully sorted queue would geometrically correspond to a line: every element is ordered with respect to all elements in front of and behind it. A partially sorted queue would similarly correspond to a cone, where there need not be any order among the many elements equidistant from the head of the queue. The queue interpretation of a heap may be confusing to Swedish readers, as Sweden is probably the only country on earth were all queues are always fully sorted.

All remaining basic operations on heaps run in time at most proportional to the height of the tree. In particular, updating a node, that is updating a $(\tau_\mu,\ \mu)$ pair, will move the node up or down the tree until the heap again satisfies the heap property. Thus, updating a node takes time at most proportional to the height of the tree, $\mathcal{O}(\log(M))$. The iterative update implementation used in `MesoRD` is given by the `resort()` function.

Function `resort(`n`)` An iterative heap-resorting, or *heapify*, algorithm for a single node. A real implementation must also handle boundary cases; a node cannot be moved off the top or the bottom of the heap. Boundary checks are omitted in the pseudo-code given here. The time of a node is queried by calling `time()`. The `maxChild()` and `minChild()` functions return references to the child nodes with the highest and the lowest times, respectively. The `parent()` function returns a reference to the parent.

```
n: the node to resort
begin
    while time(n) > time(maxChild(n)) do
        n ↔ minChild(n)
    while time(n) < time(parent(n)) do
        n ↔ parent(n)
end
```

It is not practical to store all the information about a reaction in a tree node, as these are moved around during the course of the simulation. Therefore, as mentioned in the caption of Fig. 1.3, each tree node stores a *reference* to the appropriate reaction[6]. Now, space for the reaction-related data needs to be allocated only once, and their location in memory is not rearranged by the movement of the references in the tree. Since the references in the tree are equal size, all nodes require the same amount of memory, and the space needed for the heap is not affected by node exchanges.

1.5 Coupled Reaction–Diffusion Processes

Both algorithms presented so far assume spatial homogeneity. Spatial homogeneity means that a molecule is assumed to have the same probability to interact with any of its potential reaction partners, irrespective of exactly where these are located within the reaction volume. This implies that each molecule has a high probability of diffusing through the entire volume before participating in a reaction.

It has been suggested that not even small bacterial cells demonstrate spatial, chemical homogeneity. Locally, however, the spatial homogeneity requirement may be satisfied. Such local homogeneity may arise from frequent, elastic and non-reactive collisions which tend to randomise the positions of the colliding molecules.

[6]There exists a one-to-one mapping between references, whatever their nature, and the reaction indices used in the text. If reactions are stored in static memory, indices may be interpreted as offsets from the location of the first – *zeroth* – reaction.

Local, spatial homogeneity can be exploited by dividing the total volume, V, into N subvolumes[7]. The subvolumes form a disjoint partition of V, and may therefore be uniquely identified by their location in space. Denote a subvolume located at $\mathbf{v} = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathrm{T}}$ by Ω_{xyz} or $\Omega_{\mathbf{v}}$. Thus,

$$V = \bigcup_x \bigcup_y \bigcup_z \Omega_{xyz}.$$

The subdivision of V into subvolumes is illustrated in Fig. 1.4.

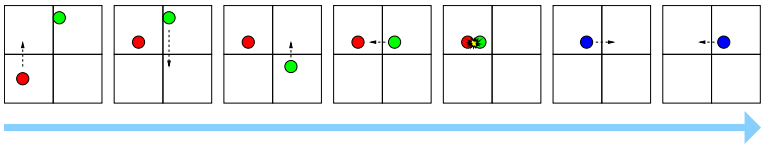


Figure 1.4: Reactions and diffusions in subvolumes. The big, two-by-two square delimits the system, which is composed of four subvolumes. The same system is shown for different points on the light blue time axis. The time between two successive states need not be equal, although the figure shows the states equidistant on the time axis. At first, there are two molecules of different species in the system. Because the red and the green molecules are in different subvolumes, they cannot react. By diffusive movement they are brought into the same subvolume. The red and the green molecules react, and are transformed into one blue molecule, which diffuses through the system. Particles may very well move by "infinitely detailed Brownian motion" between inter-subvolume jumps [19]; in the next subvolume method, the location of a molecule is known to subvolume resolution only.

The subvolumes are assumed to be spatially homogeneous. Potential reaction partners of a given molecule are now defined as all the other molecules in the same subvolume. Analogously to the two questions in the homogeneous algorithms, a non-homogeneous simulator needs to answer [12]:

1. *When and where* will the next event occur?
2. *What* kind of event will it be?

Bortz *et al.* had addressed spatial composition in two-dimensional systems in 1974 [20]. Gillespie deemed the approach computationally impractical for realistic systems, since a diffusion event would correspond to two reactions: one for destructing the molecule in the source subvolume and another for recreating it in the destination subvolume [4]. Nevertheless, in 1979 Malek-Mansour and Houard reported successful results for multi-subvolume geometries in one dimension [21].

Possible implementations in three dimensions were discussed by Hanusse and Blanché in 1981 [12]. In 1996, Stundzia and Lumsden used a Green's function to map the bulk diffusion coefficient, D, in Fick's differential law of diffusion,

[7] The subvolumes are sometimes called *cells* by physicists, which can cause confusion among biologists. They have also been called *voxels* (volume picture elements, or volume *pixels*), which may confuse computer scientists.

Eq. 1.9, to corresponding rate probabilities for diffusive events between subvolumes [6, 22]. This result enabled the formulation of the fundamental hypothesis for diffusion processes, Def. 1.5.1, similar in spirit to what was envisioned by Gillespie [4].

Definition 1.5.1 (Stundzia and Lumsden's Fundamental Hypothesis for Diffusion Processes)**.** $d_i \mathrm{d}t \equiv$ average probability, to first order in $\mathrm{d}t$, that one of ${}^{\#}s_{i\mathbf{v}}$ i:th molecules will diffuse out of the subvolume $\Omega_{\mathbf{v}}$ into an adjacent subvolume in the next time interval $\mathrm{d}t$.

Analogous to the reasoning that lead up to Eq. 1.5, one may begin the derivation by expressing the probability that neither reaction nor diffusion will occur in a subvolume in the time interval $(t,\ t+\tau)$. Considering all subvolumes in V, this means

$$P_0(\tau) = \exp\left(-\sum_{\mathbf{v}\in V}\left(\sum_{\mu=0}^{M-1} h_{\mu\mathbf{v}} c_\mu + \sum_{i=0}^{\mathcal{S}-1} {}^{\#}s_{i\mathbf{v}} d_i\right)\tau\right) \tag{1.8}$$

to first order in $\mathrm{d}\tau$.

Now the reaction–diffusion probability density function, $P(\tau,\ \mu,\ i,\ \mathbf{v})$ may be constructed. Using this function, one can find the probability for a reaction or a diffusion event occurring in the differential time interval $(t+\tau,\ t+\tau+\mathrm{d}\tau)$. Reaction and diffusion are the only two events that can change the state of the system.

- The probability of a reaction event involving R_μ in $\Omega_{\mathbf{v}}$ is $h_{\mu\mathbf{v}} c_\mu P_0(\tau)\,\mathrm{d}\tau$. Compared to Eq. 1.5, the only difference is the appearance of the location vector, $\mathbf{v}$, in the combinatorial factor $h_{\mu\mathbf{v}}$.
- The probability of a diffusion event originating in $\Omega_{\mathbf{v}}$ is ${}^{\#}s_{i\mathbf{v}} d_i P_0(\tau)\,\mathrm{d}\tau$, where the diffusing species is S_i. Each neighbouring subvolume has the same probability of being the target of the diffusion event.

These ideas, when put together, lead to Alg. 4, the coupled reaction–diffusion method. This is a method that does *not* assume spatial homogeneity. However, molecules are still treated as dimensionless point particles, which means that potentially important crowding effects are neglected [23, 24].

The columns of the state matrix $\mathbf{X}_t$ are equivalent to the state vectors from the previous algorithms. There will be one such state column vector for each subvolume. Thus, if there are $\mathcal{S}$ species and N subvolumes in the simulation, $\mathbf{X}_t$ will be a $\mathcal{S} \times N$ matrix.

As should now be apparent, the reaction-diffusion master equation has a much larger state space than its homogeneous counterpart. In the case of cubic subvolumes with six neighbours each, there are six different diffusion events for every species, and the number of events that can change the state of the system is $\mathcal{O}(N(6\mathcal{S}+R))$. In realistic systems, N is typically larger than 10^5.

1.5.1 Macroscopic and Microscopic Diffusion Constants

In a macroscopic treatment, diffusion is modelled by Brownian motion, arising from a concentration gradient [25]. Diffusion is only weakly dependant on the

Algorithm 4: The coupled reaction–diffusion method by Stundzia and Lumsden. The state matrix, $\mathbf{X}_t$, is updated in every iteration, using the diffusion parameters, $\{d_i\}_{i=0}^{\mathcal{S}-1}$, from Def. 1.5.1 and the combinatorial factors, $\{h_{\mu\mathbf{v}}\}_{\mu\mathbf{v}=0}^{MN-1}$, from Eq. 1.8. The remaining symbols are identical to those in Alg. 2. The algorithm as outlined here does not deal with determining $\mathbf{u}'$, the target subvolume in case of a diffusion event. In practise, this is not a problem, because the relative probabilities of the different diffusion targets are known. The algorithm relies on the `updateRates()` function.

begin
  `updateRates()`
  for $t \leftarrow 0$ **to** t_{stop} **do**
    $a \leftarrow \sum_{\mathbf{v}\in V} \sum_{\mu=0}^{M-1} h_{\mu\mathbf{v}} c_\mu$
    $b \leftarrow \sum_{\mathbf{v}\in V} \sum_{i=0}^{\mathcal{S}-1} \#s_{i\mathbf{v}} d_i$
    $(t,\ \mu,\ i,\ \mathbf{u}) \leftarrow (\tau_{\nu j\mathbf{v}},\ \nu,\ j,\ \mathbf{v})$ where $\tau_{\nu j\mathbf{v}} = \min_{\substack{0\le\lambda\le M-1 \\ 0\le k\le\mathcal{S} \\ \mathbf{w}\in V}} (\tau_{\lambda k\mathbf{w}})$
    if $(a+b)\cdot$ `random()` $\le a$ **then**
      Update the column of the state matrix corresponding to the subvolume at $\mathbf{u}$ to account for reaction R_μ
    else
      $\mathbf{u}' \leftarrow$ the target subvolume of the diffusion event in $\mathbf{u}$
      Update the state matrix at columns corresponding to the subvolumes at $\mathbf{u}$ and $\mathbf{u}'$ to account for diffusion of species i
    `updateRates()`
end

state of the system – the motion of one particular species is assumed to be independent of any other species.

$$\frac{\partial s_i}{\partial t} = D_i \nabla^2 s_i \tag{1.9}$$

The diffusion constant, D, in Fick's law above, in general depends on the concentration of the diffusing species.

Fick's law is deceivingly deterministic, because it describes the motion of infinitely many Brownian particles. Unsurprisingly, it breaks down in the case of low concentrations. A microscopic derivation of molecular diffusion can be achieved by using the Langevin equation, which is a statistical formulation of Newton's equation of motion. Such a derivation captures the influence of random forces on diffusing molecules [25]. The result in one dimension is similar to Eq. 1.9,

$$\frac{\partial P_i}{\partial t} = d_i \nabla^2 P_i,$$

where P_i is the distribution of the i:th species, d_i is defined by the relation

$$t_R = \frac{\langle \ell^2 \rangle - \langle \ell \rangle^2}{2d_i},$$

Function `updateRates(v)` The function, as used in Alg. 4, will update the rate constants for reaction events, $\mathbf{y}$, and diffusion events, $\mathbf{z}$, in the subvolume centred at $\mathbf{v}$. Symbols are used as defined in Alg. 4.

```
begin
    for μ ← 0 to NM − 1 do
        y_μ ← h_{μv}
    for i ← 0 to NS − 1 do
        z_i ← d_i #s_{iv}
end
```

and ℓ, in turn, is the distance a molecule will travel in time t_R. The randomness in Brownian motion implies $\langle \ell \rangle = 0$. In the context of reaction–diffusion simulation, this means that the probability for an S_i molecule to diffuse between adjacent subvolumes of size ℓ, such that $\Omega = \ell^d$, is proportional to $1/t_R = 2d_i/\left\langle \ell^2 \right\rangle$.

For the derivation to be valid, the time scale must be long enough to prevent autocorrelation in the velocities of the diffusing molecules. If the chemistry in the subvolumes is assumed to be well-defined, the autocorrelation requirement translates to $\ell \gg r_R$, where r_R is the reaction radius [26]. Intuitively, this may be understood as the lower limit on subvolume size in order to prevent reaction and diffusion from merging into an inseparable process.

On the other hand, subvolumes may not be so big as to let the initial location of a molecule within a subvolume affect the particle's reaction probability. This condition is fulfilled when [27]

$$\ell^2 \ll D t_R.$$

In summary, subvolumes should be big enough to permit nonreactive collisions to homogenise the reaction volume, but small enough for molecules to travel through a large number of subvolumes during their lifetime [2, 28].

1.6 The Next Subvolume Method

The next subvolume method assembles concepts from the previously encountered algorithms [2]. Like Stundzia and Lumsden's method, it does not require global homogeneity, as the total volume, V, is divided into N subvolumes, each with volume Ω. The reactions in each subvolume are handled by Gillespie's original algorithm. Translating ideas the from Gibson and Bruck now leads to running time $\mathcal{O}(\log(N))$ in the number of subvolumes, N, instead of $\mathcal{O}(N)$[8]. The next subvolume method is an exact method; its results are provably equivalent to trajectories of the master equation.

The algorithm now explained is assembled in Alg. 7. In the next subvolume method, the *subvolumes*, not the reactions, are stored in a heap, A. This crucial step is what sets the next subvolume method apart from many other methods [29]. The subvolumes are initialised, meaning that for each subvolume, one computes and stores the sum of the reaction and diffusion rates, as

[8]The complexity is $\mathcal{O}(\log(N))$ only as long as the number of subvolumes is much larger than the number of reactions, species *and* neighbours per subvolume. This is normally the case.

well as the time until the next event. These computations are handled by the `updateSubvolume()` function. The subvolumes are then sorted with respect to the previously computed time, using a standard sorting algorithm. The sorting will make sure that the subvolumes satisfy the heap property: if $A[i-1] \leq A[i]$ one trivially has $A[\lfloor i/2 \rfloor] \leq A[i]$, where $1 \leq i \leq (N-1)$.

Function `updateSubvolume(`*i, t*`)` The function will update the rates for reaction and diffusion events in the i:th subvolume, using a single random number. The number of neighbours of the i:th subvolume is denoted by n_i, and the size of any subvolume is ℓ. The function will also update the time of the next event in the subvolume, using the current simulation time, t. Remaining symbols are used as in Alg. 4.

---

i: index of the subvolume to update
t: the current simulation time
begin

$$\rho_i \leftarrow \sum_{\nu=0}^{M-1} a_\nu(\mathbf{X}_t) \text{ where } a_\mu(\mathbf{X}_t) = h_{\mu i} c_\mu$$

$$\delta_i \leftarrow \sum_{j=0}^{\mathcal{S}-1} \sum_{\lambda=0}^{n_i-1} \frac{d_{j\lambda} \mathbf{X}_{ij}}{\ell^2}$$

$$\tau_i \leftarrow \frac{1}{\rho_i + \delta_i} \cdot \ln\left(\frac{1}{\texttt{random()}}\right) + t$$

end

In the time loop, the next subvolume is found. This is done in constant time, as the next subvolume is always referred to by $A[0]$. A random number is used to find whether the event in the next subvolume will be a reaction or a diffusion event.

- In the case of a reaction event, the same random number can be reused to determine which reaction will occur. After executing the reaction, the subvolume has to be updated: the total reaction and diffusion rates and the time until the next event are recomputed. This will require a second random number. The subvolume may need to be reordered in the heap.
- In the case of a diffusion event, the first random number can be reused in order to determine which species will diffuse, and which of the neighbours will be the target of the diffusion event. After executing the diffusion, both source and target subvolumes need to be updated and possibly reordered in the heap. This will consume two random numbers, one for each subvolume.

Note that one only needs to update rates in subvolumes where the composition has changed. Making use of this observation has roughly the same effect as the dependency graph in Gibson and Bruck's method[9]. Thus, if the last event was a reaction event, there will be one subvolume to update, if it was a diffusion event, there will be two. Reestablishing the heap property is an $\mathcal{O}(\log(N))$ operation, and this has to be done for each modified subvolume.

While updating ρ_i and δ_i, it would be possible to determine not only *when* the next event will occur, but also *what* kind of event it will be. The advantage of

[9]Some implementation details regarding dependency graphs for this particular purpose are discussed by Andér [30].

such an approach is that all calculations on a subvolume are handled in a single pass. Such an arrangement would facilitate parallel implementation, but comes at the price of unnecessary computation. Since the composition of a subvolume may change again *before* the next event occurs in the subvolume, any of these calculations on the subvolume would be invalidated. Because diffusion events are typically much more frequent than reaction events, it is not unlikely that the state of a subvolume will change before the next event occurs in it. The same type of problem occurred in the first-reaction method. There, it was solved by introducing a dependency graph.

The reaction-dependency graph from Gibson and Bruck's method is currently not implemented in the next subvolume method. This decision was motivated by memory considerations. If one wants to avoid recalculation of reaction rates, one must obviously store those rates that need *not* be recalculated. Since the $\{a_\mu\}$ are unique for every subvolume, this would require the storage of $N \cdot M$ additional floating point values. For big systems, N and M are large, and memory consumption could become a prohibiting factor. It may however be interesting to implement a dependency graph in a later version of `MesoRD`, preferably as a compile-time or modular option.

Algorithm 7: Elf's next subvolume method. The algorithm uses two random numbers per iteration in the case of reaction events and three random numbers per iteration for diffusion events. The difference between $\{d_j\}$ as discussed in the text, and the $\{d_{jk}\}$ used in the algorithm, is that the latter respect boundary conditions. For instance, if the subvolume under consideration does not have a neighbour in the k:th direction, one would have $d_{jk} = 0$, irrespective of the diffusing species index j. The number of neighbours of the i:th subvolume is denoted by n_i, and the size of any subvolume is ℓ. The remaining symbols are identical to those in Alg. 4. The algorithm as given here relies on the `updateSubvolume()` function.

begin
- **for** $i = 0$ **to** $N - 1$ **do**
  - `updateSubvolume(`i`)`
- Sort all the subvolumes in order of increasing τ_i
- **while** $t < t_{stop}$ **do**
  - $(t,\ i) \leftarrow (\tau_j,\ j)$ where $\tau_j = \min_{0 \leq k \leq N-1} (\tau_k)$
  - $\mathcal{R} \leftarrow$ `random()`
  - **if** $\mathcal{R} < \frac{\rho_i}{\rho_i + \delta_i}$ **then**
    - $\mathcal{R} \leftarrow \mathcal{R} \cdot \frac{\rho_i + \delta_i}{\rho_i}$
    - $\mu \leftarrow \lambda$ such that $\sum_{\nu=0}^{\lambda-1} a_\nu (\mathbf{X}_t) < \mathcal{R} \sum_{\nu=0}^{M-1} a_\nu (\mathbf{X}_t) \leq \sum_{\nu=0}^{\lambda} a_\nu (\mathbf{X}_t)$
    - Update the state matrix $\mathbf{X}_t$ to account for reaction R_μ
    - `updateSubvolume(`i`)`
    - `resort(`i`)`
  - **else**
    - $\mathcal{R} \leftarrow (\mathcal{R} - \rho_i) \frac{\rho_i + \delta_i}{\delta_i}$
    - $(j,\ k) \leftarrow (l,\ m)$ such that
      $$\sum_{p=q=0}^{p=l-1,\ q=m-1} f_{pq} < \mathcal{R} \sum_{p=q=0}^{p=\mathcal{S}-1,\ q=n_i-1} f_{pq} \leq \sum_{p=q=0}^{p=l,\ q=m} f_{pq}$$
      where $f_{pq} = \frac{d_{pq} \mathbf{X}_{pi}}{\ell^2}$
    - Update the state matrix $\mathbf{X}_t$ to account for diffusion of species k from subvolume i to subvolume j
    - **for** $k \in \{i,\ j\}$ **do**
      - `updateSubvolume(`k`)`
      - `resort(`k`)`

end

Chapter 2

An Overview of MesoRD

2.1 Running MesoRD

A simulation with MesoRD progresses through two distinct phases.

- In the first phase, the model definition is read from an existing file. Models are defined using the Systems Biology Markup Language, SBML [31], an extended markup language discussed further in section 2.2. MesoRD will build an internal representation from the information in the file. The internal representation in terms of the MesoRD implementation is summarised in section 2.3.
- In the second phase, the system is simulated. This means that Alg. 7 is applied to the data gathered in the previous step. During simulation, the need to interact with the user may arise. The threading that enables such interaction to occur in parallel with simulation is discussed in section 2.5.

The first phase is, in general, neither difficult nor computationally demanding. However, due to the expressiveness of SBML, several thousand lines of code are devoted to the construction of the internal representation. The performance of the first phase is almost exclusively dependent on the complexity of the system's geometry. Geometry generation is the subject of chapter 3; large or complex geometries may take considerable time to build.

The second phase does the actual simulation work. Only a small fraction of the code is concerned with this phase of the program – yet, as it is an iterative procedure, it can, depending on user settings, consume an almost arbitrary amount of time. The small size and low complexity of the simulator code means that it is relatively easy to maintain. Improvements and optimisations in the simulation engine can have a tremendous impact on overall performance.

2.1.1 Errors and Warnings

The only errors of concern to end users are those that can be addressed without modifying the MesoRD source code. Because the user's only responsibility is to supply MesoRD with a model definition and suitable simulation parameters, any errors arising from user mistakes can be detected and reported during the model building phase. Simulation performance need not – in fact, *should not* –

be burdened by input validation or error checking. Once the internal model is correctly built and simulation starts, any subsequent errors will be due to bugs, which, in a perfect world, the end user should not need to worry about.

Errors in the SBML model definition are reported by the `libsbml` library [32]. `MesoRD` itself does no error reporting whatsoever, and modelling mistakes not detected by `libsbml` cause the program to crash silently. Because `MesoRD` introduces several extensions to SBML, which `libsbml` knows nothing about and, hence, cannot check for correctness, this is a significant shortcoming. The lack of error reporting makes usage of `MesoRD` prohibitively difficult for non-developers.

Coping with errors will probably be implemented as a globally accessible list of messages. Whenever an error is encountered, a message is appended to the list. The list is checked and reported *after* model building but *before* invoking the simulation algorithm. This is similar to the strategy implemented by proper compilers, which report as many errors as they can before terminating.

2.2 The Model

Details of model building using SBML are discussed elsewhere [33]. SBML is *not* meant to be authored manually; its primary purpose is to serve as a standard for representation, storage and exchange of biochemical models. Editing is supposed to be done through model building software. Because modifications and extensions to the original language have been necessary in order to define models suitable for simulation in `MesoRD`, there are no third-party programs that can build `MesoRD` models out of the box.

The syntax of `MesoRD`'s SBML dialect is discussed in the User's Guide included in the software distribution [34]. The internal mechanism by which extensions are dealt with is reviewed in section 2.2.2. Note also that current versions of `MesoRD` do not implement *all* features provided by SBML.

2.2.1 The Systems Biology Markup Language

In brief, an SBML file is structured as shown in Fig. 2.1. The stacked `listOfX` elements contain one or more stacked definitions of `X`. Any of these definition lists can be empty, in which case the corresponding `listOfX` construct need not appear at all. The order of `listOfX` constructs is not important.

In `MesoRD`, the opening and closing `listOfX` tags mark the boundaries of separate namespaces. There could, for instance, be both a unit and a parameter named `foo`. The accessible namespaces at any point are context-dependent: the definition of a species may refer to a compartment, but the definition of a unit can never refer to anything but other units. Thus, the compartment namespace is visible from the species namespace but is inaccessible from the unit namespace. Some definitions can be nested: for example, a reaction can define its own set of parameters, which may shadow any globally defined parameters. The scope of locally defined parameters only extends to the particular reaction.

`MesoRD`'s scoping and namespacing has a few implications on model building. Because a reaction definition may refer to parameters as well as species, the parameter namespace as well as the species namespace must be accessible from the reaction definition. Under these circumstances, it is not safe to use the same

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2"
      level="2" version="1">
  <model id="My_Model">
    <listOfEvents>
      ...
    </listOfEvents>
    <listOfFunctions>
      ...
    </listOfFunctions>
    <listOfUnitDefinitions>
      ...
    </listOfUnitDefinitions>
    <listOfCompartments>
      ...
    </listOfCompartments>
    <listOfSpecies>
      ...
    </listOfSpecies>
    <listOfParameters>
      ...
    </listOfParameters>
    <listOfRules>
      ...
    </listOfRules>
    <listOfReactions>
      ...
    </listOfReactions>
  </model>
</sbml>
```

Figure 2.1: A skeleton SBML file [33]. SBML is an extended markup language, XML, which means that it is hierarchical in structure and easily readable by machines as well as humans. In principle, an SBML file contains a stacked list of stacked definitions. Note that events, functions and rules are not yet supported in `MesoRD`.

identifier in different namespaces[1]. In fact, some of these uses are not even valid SBML. Exploiting `MesoRD`'s peculiar scoping and namespacing feature is discouraged, even in cases where the outcome is well defined.

The dependencies among the different components of an SBML model dictate a certain order in the building of the internal representation. For instance, building an internal representation of the reactions requires knowledge about the reacting species. Therefore, the species must be internalised before the reactions. Units may be needed in almost any part of the model. Thus, units have to be parsed at the very beginning. A flow chart of the model building is shown in Fig. 2.2.

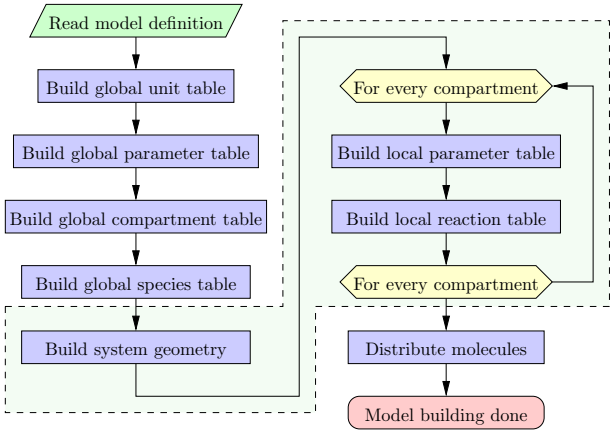


Figure 2.2: A flow chart of the building of the internal representation from an SBML model. The flow chart is implemented in the `fromSBML()` function. The shaded background indicates that the system geometry, as described in chapter 3, is built together with the compartments.

2.2.2 Annotations

The extensions to SBML, required for simulation in `MesoRD`, are supplied in SBML annotations. Any SBML element can carry an annotation, but `MesoRD` will ignore any annotations it cannot handle. Annotations are defined in XML, using a namespace different from the one used for the standard SBML elements. Annotations in `MesoRD` are parsed and accessed using the `Annotation` class.

The `Annotation` class provides functions for accessing and comparing tag names, child nodes and attribute name–value pairs of XML nodes. The class relies on the SAX API of the either the `Expat` [35] or the `Xerces-C++` library [36] for the actual parsing. The `Annotation` class also implements a set of exceptions to deal with possible run-time errors. For example, in order to obtain the single-precision floating point value of an attribute `foo`, one would

[1] Assume that one has defined a parameter as well as a species identified by `foo`. Furthermore assume the definition of a reaction with rate law `bar` · `foo`, where `bar` is a parameter. It is then not at all apparent whether `foo` refers to a species or a parameter. If the model writer has supplied units wherever possible, `MesoRD` *could* conduct a dimension analysis in order to resolve the referred entity, since one knows that a rate law should evaluate to $\mathtt{s}^{-1}$. However, because use of units is not compulsory in SBML, this sort of deduction is generally impossible.

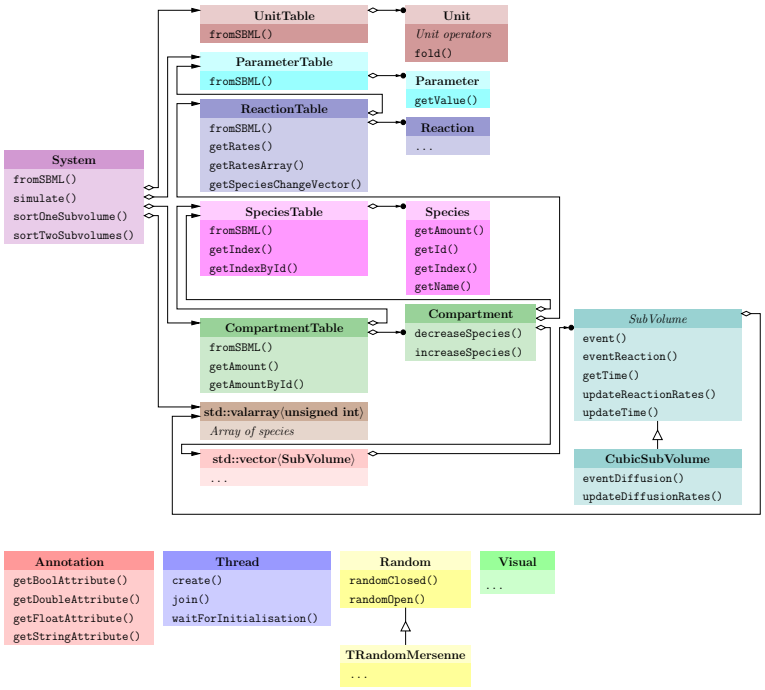


Figure 2.3: Class layout for `MesoRD`, including some auxiliary classes like `Annotation` and `Thread`. In addition to the classes shown and discussed here, the code contains several others that assist or complement the ones depicted above. For instance, the figure shows no classes for exception handling, and the CSG and (G)UI modules are omitted entirely. For some classes, a few key functions are indicated; the real classes have many more. The figure loosely adheres to the object modelling technique, OMT, as adapted by Gamma *et al.* [38]: arrows show aggregation, connecting triangles symbolise inheritance. When the name of a class is set in italics, it indicates that the class is abstract.

call `getFloatAttribute("foo")`. In case the value cannot be converted to a `float`, an `InvalidFormat` exception will be thrown. The functions for obtaining double-precision floating point, string and Boolean values are similar.

2.3 Class Overview

`MesoRD` is an object oriented program, written in C++. Good C++ programming practise states that major concepts should correspond to classes, and that related concepts should share a common interface [37]. An overview of the relationship among the classes involved in the internal representation of the model is given in Fig. 2.3. More detailed information about the `MesoRD` class layout is given in the source-level documentation [34].

Building an internal representation of a model amounts to establishing and keeping track of, possibly nested, identifiers, or *symbols*, and their associated values. Just like an ordinary compiler that faces the same task, `MesoRD` imple-

ments symbol tables to map identifiers to values. Comparing Figs. 2.1 and 2.3, one may notice that each supported `listOfX` construct in SBML corresponds to a `XTable` class in the `MesoRD` implementation. These classes implement the symbol tables for symbols of type `X`.

The purpose of the `XTable` classes is to allow access to definitions by some identifier, which has to be unique in the current scope and namespace. For most elements, the SBML `id` field acts as identifier. The notable exception is the species, which are identified by their SBML `name` field. The reason for this is further discussed in section 2.3.4.

2.3.1 The `System` and `SubVolume` Classes

The central classes that implement the simulation algorithm are `System` and `SubVolume`. The `System` class contains symbol tables for the compartments, parameters and units used in the model. It also maintains the heap of all the subvolumes that comprise the system's geometry. The functions that populate the tables and build the geometry, given an SBML file, are invoked by calling `fromSBML()`. Once the internal representation of the model is built, the heap is initialised by computing the time until the next event in every subvolume. This is accomplished through the `updateTime()` function. Finally, the system is simulated using the `simulate()` function. The `simulate()` function in turn needs to call various functions from the `SubVolume` class.

As was mentioned in section 1.2, the state of the system is defined by the location of every single molecule. The spatial location of a molecule is given by the coordinates of the subvolume in which it is found[2]. The union of the states of all subvolumes is therefore equivalent to the state of the entire system. The state of each subvolume, in turn, is maintained in the respective instances of the `SubVolume` class. Maintaining state for a subvolume boils down to knowing the copy number of each molecular species. In practise this is done by storing the number of each species in a state vector, as is further discussed in section 2.3.4.

The state of a subvolume can be accessed and modified by various getter and setter functions. Changing the state of a subvolume requires the time for the next event to be updated. Time updates are performed in three calls: invoking `updateDiffusionRates()` and `updateReactionRates()` will update δ and ρ as used in Alg. 7. These computations do not require any random numbers. Once δ and ρ again reflect the current state of the subvolume, calling `updateTime()` will, using a single random number, determine the time until the next event in the subvolume. The time for the next event is retrieved by calling `getTime()`.

Reaction events depend on the volume, but not the the geometry, of the subvolumes in which they take place. Diffusion events depend on inter-subvolume distances. Thus, if ℓ describes the size of the subvolume, subvolume geometry is a factor in determining $\delta = \delta(\ell)$, but not ρ. `MesoRD` was designed to support several kinds of subvolume geometries. Therefore, the `SubVolume` class is an abstract class, providing an implementation only for change of state and reactions. It also defines an interface for diffusion functions. `CubicSubVolume`,

[2]Unlike in `Smoldyn`, the exact location of a molecule within a subvolume is unknown to `MesoRD` [39]. Thus, spatial pinpointing of molecules can be done to subvolume resolution only. In fact, given the time resolution of the method, this is the most accurate pinpointing possible.

a class derived from `SubVolume`, implements the particularities relating to diffusion in the special case of cubic subvolumes of a given side length, ℓ. In the future, `MesoRD` may feature other classes derived from the `SubVolume` class, implementing different kinds of subvolumes.

In order to execute the next event, one should call the `event()` function. This function will, using a fresh random number and the previously computed ρ and δ, find and execute the next event. The `event()` function returns a reference to the subvolume that, in addition to the subvolume in which the event took place, has been modified. Since reactions only affect the current subvolume, `event()` will return `null` in case of reaction events. Internally, the `event()` function calls either `eventDiffusion()` or `eventReaction()` in order to execute a diffusion or a reaction event respectively.

Any subvolume which has had its state, and therefore also its time, changed due to the executed event may need to be reordered in the heap, so that the heap property remains satisfied. As was mentioned in section 1.4, reestablishing the heap property means moving a reference to the changed subvolume up or down the tree representation of the heap. The time taken for such an operation scales linearly with the displacement distance. In the worst case, this is the full height of the tree, $\mathcal{O}(\log(N))$. It is possible to estimate the actual displacement distance of a subvolume by investigating the difference $\Delta\tau = \tau_{new} - \tau_{old}$. A negative sign means the subvolume will move towards the root, a positive sign will force the subvolume towards the leaves, see the `resort()` function outlined in section 1.4.1. The magnitude of $\Delta\tau$ may serve as an heuristic approximation of the distance a subvolume will need to move.

The `System` class in `MesoRD` defines two functions for subvolume reordering: `sortOneSubVolume()` and `sortTwoSubVolumes()`. The reason for separate functions for sorting one and two subvolumes is that performance can be slightly improved in the case of two modified subvolumes: one may, prior to resorting the individual subvolumes, exchange their locations in the heap if that would decrease the total magnitude of $\Delta\tau$. The operation still has formal complexity $\mathcal{O}(\log(N))$, where N is the number of subvolumes, but the coefficient has decreased.

2.3.2 Units

Units are essential to any computation in kinetics; if the mesoscopic rate constant for a reaction is 22.17, it matters a great deal if the number is to be interpreted in s^{-1} or $\mathrm{mole} \cdot \mathrm{s}^{-1}$. SBML supports a wide range of units for this purpose. Furthermore, SBML gives the user the possibility to define new units, composed of any number of pre-defined units. The operations on units in `MesoRD` are implemented in the `Unit` class. The system holds a global symbol table for units, defined in the `UnitTable` class.

Internally, `MesoRD` only uses basic and supplementary SI units [40]. Any unit, whether user-defined or defined by SBML, is converted to a, possibly scaled and offset, product of SI units. Thus, one could for instance define a unit `fulmeter` as $\mathrm{J} \cdot \mathrm{N}^{-1}$, and subsequently specify all lengths in `fulmeter`.

The construction of a `MesoRD` unit from an SBML definition proceeds through two steps: first the unit is reduced to a product of SI-units, then the product is algebraically simplified, or *folded*. The reduction is done while the unit is read from the SBML definition; only the parts that are concerned with parsing

SBML need to know about the plethora of SBML units. The folding is taken care of by the `fold()` function.

The process for the `fulmeter` unit is illustrated in Eq. 2.1: the original unit definition is given in Eq. 2.1a, the reduction step is shown in Eq. 2.1b and the final folding step is done in Eq. 2.1c.

$$\begin{aligned} \texttt{fulmeter} &= \frac{\mathrm{J}}{\mathrm{N}} = && (2.1a) \\ &= \frac{\mathrm{kg} \cdot \mathrm{m} \cdot \mathrm{m/s^2}}{\mathrm{kg} \cdot \mathrm{m/s^2}} = && (2.1b) \\ &= \mathrm{m} && (2.1c) \end{aligned}$$

Units in `MesoRD` can be added, subtracted, multiplied and divided. They can also be raised to some power and tested for equality. The operations are implemented using C++ overloading of the standard operators `+`, `-`, `*`, `/`, `^`, `==` and `!=`. Conversion factors may be necessary in order to perform an actual operation with values of different units. For example:

$$1 \text{ item} + 1 \text{ mole} = 1 \text{ item} + N_A \text{ item} = (1 + N_A) \text{ item},$$

where N_A is Avogadro's number.

- If two units are *compatible*, they can be added or subtracted. For instance, m and °C are not compatible, and attempting to add those units would throw an exception. Adding m and km, or °C and K is permitted. Performing the latter addition will also involve conversion factors.
- Multiplying and dividing two units is *always* permitted. For instance, dividing m by °C will result in the unit $\mathrm{m} \cdot {}^\circ\mathrm{C}^{-1}$. Similarly, raising K to the power of -2 yields K^{-2}. Multiplying m by km results in $10^3 \cdot \mathrm{m}^2$.
- There is no immediately obvious way to implement equality of units. For instance, should m equal km? `MesoRD` currently implements the view that two units are equal *if and only if* they can be added *without* applying any scaling factors or offsets. Thus, there is a difference between *compatible* units and *equal* units: `MesoRD` regards m and km as unequal, even though they can be added, and therefore are compatible.

The Trouble with Offset Units

Offset units are problematic[3]. They are also fairly uncommon. One of the few sensible examples involves the conversion between °C and K:

$$\begin{cases} T_C = T_K - 273.15 \\ T_K = T_C + 273.15 \end{cases} \quad \text{where } T_C \text{ has unit °C and } T_K \text{ has unit K.}$$

[3]This issue has been the subject of discussion in the SBML community. Future versions of SBML, perhaps even level 2 of version 2, currently in preparation, may remove offset units altogether.

Consider for example a quantity given in $^\circ\mathrm{C} \cdot \mathrm{m}$. If one wanted to add something given in $^\circ\mathrm{C} \cdot \mathrm{m}$ to something else given in $\mathrm{K} \cdot \mathrm{m}$, one would need to do a conversion. In most cases, this is not only tricky, but impossible.

Assume the first quantity was obtained by multiplying a °C with b m. The numbers a and b are unknown, only their product ab, with unit $^\circ\mathrm{C} \cdot \mathrm{m}$, is known. The value in Km is $(a + 273.15) \cdot b = ab + 273.15 \cdot b$. This value cannot be computed unless both a and b are known. Thus, the conversion is impossible.

`MesoRD` tries to solve as much of the problem as possible:

- If two units, each of which is composed of a single, basic or supplementary SI unit, are, or can be made, equal up to an offset, they are compatible. For instance, it is always possible to inter-convert °C and K.
- If two units, each composed of arbitrarily many basic or supplementary SI units, are, or can be made, equal up to any number of offset factors, all of which have *negative* exponents, the offsets are ignored. For instance, $\mathrm{m} \cdot {}^\circ\mathrm{C}^{-1}$ is equivalent to $\mathrm{m} \cdot \mathrm{K}^{-1}$.

2.3.3 Parameters

A parameter in SBML associates a quantity with a symbolic name. The symbols may be used in mathematical formulae which in turn can define kinetic laws that control reactions. Note in particular that reactions can have local parameters, complementing or shadowing any globally defined parameters. Thus, parameter tables are nested, and the root parameter table contains only those parameters defined global to the system.

Suppose `MesoRD` is about to evaluate the rate of a particular reaction involving parameters. In order to complete the evaluation, the evaluator must retrieve the value associated with the parameter's identifier. First, the reaction's local parameter table is queried. If the requested symbol is not found, the evaluator will recursively query the parent tables until the root parameter table is reached. As soon as the symbol is found, its value is returned, and evaluation can proceed. If the query fails even for the root parameter table, there must be an error in the model, because it references an undefined, or out-of-scope parameter.

2.3.4 Species

SBML defines species as substances or entities that take part in one or more reactions. Initial species composition can be specified in terms of concentrations or absolute numbers of molecules. Internally, `MesoRD` works with absolute copy numbers, so the initial species composition is transformed to a set of integer numbers of molecules, no matter how it was specified in the model. If the initial amount of a species was specified using a concentration, `MesoRD` will use the appropriate compartment volume to convert and round the concentration into a non-negative integer species count.

In `MesoRD`, molecules of a particular species are free to diffuse between neighbouring subvolumes. This movement is controlled by diffusion rate constants, defined in annotations of the species definitions. Diffusion between subvolumes of *different* compartments causes complications, because the diffusive environment may vary with the compartment. In `MesoRD`, every species definition must

have a diffusion constant for movement to any other reachable compartment. Thus, if there are $\mathcal{C}$ compartments, each of which contains $\mathcal{S}$ species, the model will need to define $\mathcal{C}^2\mathcal{S}$ diffusion constants. Any undefined diffusion constants are assumed to be zero, meaning the probability of a diffusion event of the species between the compartments under consideration is zero.

Since a diffusion constant is not only tied to a particular species, but also to a particular combination of source and target compartments, the rate of diffusion from one compartment to another need not equal the rate in the opposite direction. This can be exploited to, for instance, construct crude models of facilitated transport.

Species Storage

All instances that need to keep track of species do so in vectors, or linear arrays. Every instance of the `SpeciesTable` class holds one such vector, as does every instance of the `SubVolume` class. In the case of subvolumes, the vector is equivalent to the state vector from chapter 1. The vectors are indexed by integer numbers, not species identifiers or species names. Thus, it is impossible to ask a subvolume for the amount of a species identified by a textual identifier. It is, however, possible to ask for the number in the i:th element of the vector. The mapping between species identifiers and indices is handled by the `SpeciesTable` class; the relevant functions are `getIndex()` and `getIndexById()`.

Referring to species by integer indices rather than character strings of arbitrary length, makes look-up operations cheap. Furthermore, memory consumption is reduced because the vectors need not store any information that would aid in index mapping[4]. In the case of subvolumes, the state vector comprises the *only* memory dedicated to species management. Because the number of subvolumes in realistically sized systems can be large, it is important to minimise the memory required to keep the state of each subvolume.

Multi-compartment Species

SBML definitions of species are tied to compartments; species that occur in several compartments must be defined several times[5]. Each definition must have a unique `id`. This means that there could be several definitions, and thus several identifiers, that refer to the same species.

Multi-compartment species in `MesoRD` are united by `name`. Assume that a system has two compartments, `C1` and `C2`, and one species, named `A`. One would need to define `A` twice, once for each compartment. Each definition, however, must have a unique `id`. Let the definition of `A` in `C1` be `A1` and similarly `A2` in `C2`. Thus, if one wanted to know how many `A` molecules are currently contained in compartment `C1`, one would call `getAmountByName("A")` from the `SpeciesTable` instance of compartment `C1`. This would yield the same result as calling `getAmountById("A1" )` from `C1`. Calling `getAmountById("A2")` in compartment `C1` would yield an error.

[4]Model writers have the opportunity to waste arbitrary amounts of memory by defining species that are never used. Each redundant species wastes $\mathcal{O}(N_c)$ of storage, where N_c is the number of subvolumes in the compartment where the species is defined.

[5]Apparently, this is awkward not only for `MesoRD`. Future revisions of SBML may enable more elegant approaches for dealing with multi-compartment species.

2.3.5 Reactions

An SBML reaction is a statement describing some transformation, transport or binding process that can change the amount of one or more species in a subvolume. While the transport aspect of reactions would allow diffusion to be treated as an SBML reaction, `MesoRD` uses reactions *only* to represent the kind of chemical events which transform reactants into products. Reactions have associated kinetic rate expressions describing how quickly they take place. Rate expressions and their evaluation is the topic of chapter 4.

Since reactants and products are species, which by their identifier are tied to a unique compartment, reactions are, by implication, also tied to the same compartment. Defining reactions that mix species from different compartments is an error. Reactions in `MesoRD` are *unidirectional*; in order to simulate reversible reactions, the model writer has to define two reactions, one in each direction.

As was mentioned in section 2.3.4, `MesoRD` internally references species by integer indices. The way a reaction changes the composition in the subvolume where it takes place is described by a *change vector*, $\mathbf{x}'$. For each species index, the corresponding element of the change vector gives the change that occurs to the species population when the particular reaction is executed. To illustrate this, consider a simple 5-species system. A reaction in which one molecule of species 2 reacts with one molecule of species 4 to produce two molecules of species 1,

$$S_2 + S_4 \longrightarrow 2S_1,$$

has the change vector

$$\mathbf{x}' = \begin{bmatrix} 0 & 2 & -1 & 0 & -1 \end{bmatrix}^{\mathrm{T}}.$$

Note again that the index of the first element in the vector is 0, not 1. The change vector of a reaction is accessed by calling `getSpeciesChangeVector()` in the `ReactionTable` class.

In order to evaluate the rates of any or all reactions in a particular subvolume, the `ReactionTable` class provides the `getRates()` and `getRatesArray()` functions. Because reaction rates generally depend on the subvolume's current species composition, both functions take a state vector as an argument.

Reaction Rate Hashing

Reaction rate evaluation can be expensive. It is also a frequent operation – rate evaluation has to be performed every time the composition of a subvolume changes. Furthermore, it is assumed that a typical simulation will evaluate reaction rates for the same configuration of species many times. In order to reduce the cost of frequent, but identical, evaluations, `MesoRD`'s compartments use hash tables to store previously computed rates.

When a reaction is about to be evaluated, `MesoRD` computes a hash value based on the configuration, using a hash function, $h(\mathbf{x})$, as defined in Eq. 2.2. The argument of the hash function is the current $\mathcal{S}$-dimensional species configuration vector, $\mathbf{x}$.

$$h(\mathbf{x}) = \left(\sum_{i=0}^{\mathcal{S}-1} 2^i x_i \right) \mod h_s, \tag{2.2}$$

where the size of the hash table is denoted by h_s.

If the hash table has an entry at $j = h(\mathbf{x})$, it consists of a configuration vector, $\mathbf{x}_{\text{hash}_j}$, and a rate for each reaction, stored in a vector $\mathbf{r}_{\text{hash}_j}$. The configuration $\mathbf{x}$ is compared to the configuration of the stored entry, $\mathbf{x}_{\text{hash}_j}$. If the configurations match, the rates will be those stored in $\mathbf{r}_{\text{hash}_j}$. The rates need not be evaluated anew.

Alarmingly little thought went into the construction of the hash function; improvements may be implemented in future versions. Furthermore, no profiling analysis of `MesoRD` has been undertaken. If reaction rate hashing yields any performance improvement at all remains to be determined.

2.3.6 Compartments

In SBML, a compartment is a container of finite size for substances. Compartments do not necessarily have to correspond to actual structures inside or outside a cell. The most important extension to compartments in `MesoRD`, is that every compartment needs a three-dimensional geometry, see chapter 3. Operations on compartments in `MesoRD` are implemented in the `Compartment` class; the symbol table for compartments is defined in the `CompartmentTable` class.

The system contains one `CompartmentTable`, which holds references to all compartments in the system. Every compartment can be queried in order to find its volume and the diffusion constants for any diffusion event originating in the compartment. A compartment holds a `ReactionTable` and a `SpeciesTable`, where the former enumerates the reactions that can act on the species contained in the latter. The `Compartment` class also defines functions to uniformly distribute species to all subvolumes of the compartment. These functions are only used during initialisation.

At all times, the compartment's `SpeciesTable` holds the total amount of every species found in the compartment. This enables semi-global statistics to be kept efficiently at the expense of having to mirror each change in a subvolume's composition in its compartment's species table as well. For instance, assume that compartment C_k defines a reaction R_μ,

$$S_i \longrightarrow S_j.$$

On executing this reaction, the amount of S_i should decrease by one, and the amount of S_j should increase by one in the subvolume where the reaction occurred. In order to keep the compartment's statistics current, the amounts in the compartment's species table must be updated as well. Statistics are output using various check-pointing functions defined in the `CompartmentTable` class.

2.4 Loose Ends

2.4.1 Random Numbers

As was seen in chapter 1, any stochastic simulation relies on a pseudo-random number generator, *PRNG*. Random numbers can be drawn from a uniform distribution using the `rand()` function from the standard `C` library. Its limited randomness, however, could lead to troublesome computational inaccuracy [4].

Some operating systems also include a `random()` function, with a significantly larger period than the historic `rand()` call. `MesoRD`'s `Random` class uses `random()` to supply the simulation algorithm with random numbers. A random number in the closed interval [0, 1] can be obtained by the `randomClosed()` function. Similarly, a random number in the open interval (0, 1) is given by invoking the `randomOpen()` function.

The quality of the native PRNG varies considerably with the particular operating system involved. The strain `MesoRD` puts on the PRNG far exceeds the capabilities of certain systems. For this reason, `MesoRD` comes with its own PRNG. The `TRandomMersenne` class, a class derived from `Random`, implements the Mersenne Twister [41]. The Mersenne Twister is a modern, reasonably fast, 623-dimensionally, equidistributed, uniform PRNG, with a period of $2^{19937} - 1$.

2.4.2 Viewer

`MesoRD` includes a simple real-time viewer. The viewer accesses the system class' list of subvolumes, including any molecules contained therein, and renders them in 3D. The viewer is not crucial for simulation; it is possible to build and run `MesoRD` on Unix-like systems which completely lack the X Window System.

The viewer is implemented in the `Visual` class. The next to trivial rendering implementation relies on OpenGL [42, 43]. The bulk of the code in the viewer is concerned with the particularities of different windowing systems. Because `MesoRD` needs more control over the viewer than is offered by toolkits such as `GLUT`, `MesoRD` rolls its own window management.

Currently, the `MesoRD` viewer is supported on the X Window System and Microsoft Windows. The X Window System implementation relies on GLX for low-level window management and the associated event processing [44]. On Microsoft Windows, the native API is used. Note that recent versions of Mac OS X are shipped with XFree86, a free X Window System implementation.

2.5 Threading and User Interaction

Once `MesoRD` has built an internal representation of the model, simulation starts. Optionally, `MesoRD` will launch the viewer and a progress bar. The viewer was discussed briefly in section 2.4.2. The progress bar, besides displaying the progress of the simulation, periodically samples input from the user, and allows for graceful termination of the simulation, without risking corruption of the produced data.

`MesoRD` must thus manage three tasks simultaneously. While simulation performance is of prime concern, the viewer would not be real-time if it did not to receive a fair share of CPU attention. The progress bar is, in comparison, computationally undemanding. In order to allow for efficient switching between these three tasks, `MesoRD` allocates each to its own thread. The life and death of these threads is illustrated in Fig. 2.4.

`MesoRD` uses preemtible POSIX threads on systems that support them [45]. On Microsoft Windows, `MesoRD` uses the native threads API. Threads-related issues are dealt with in the `Thread` class.

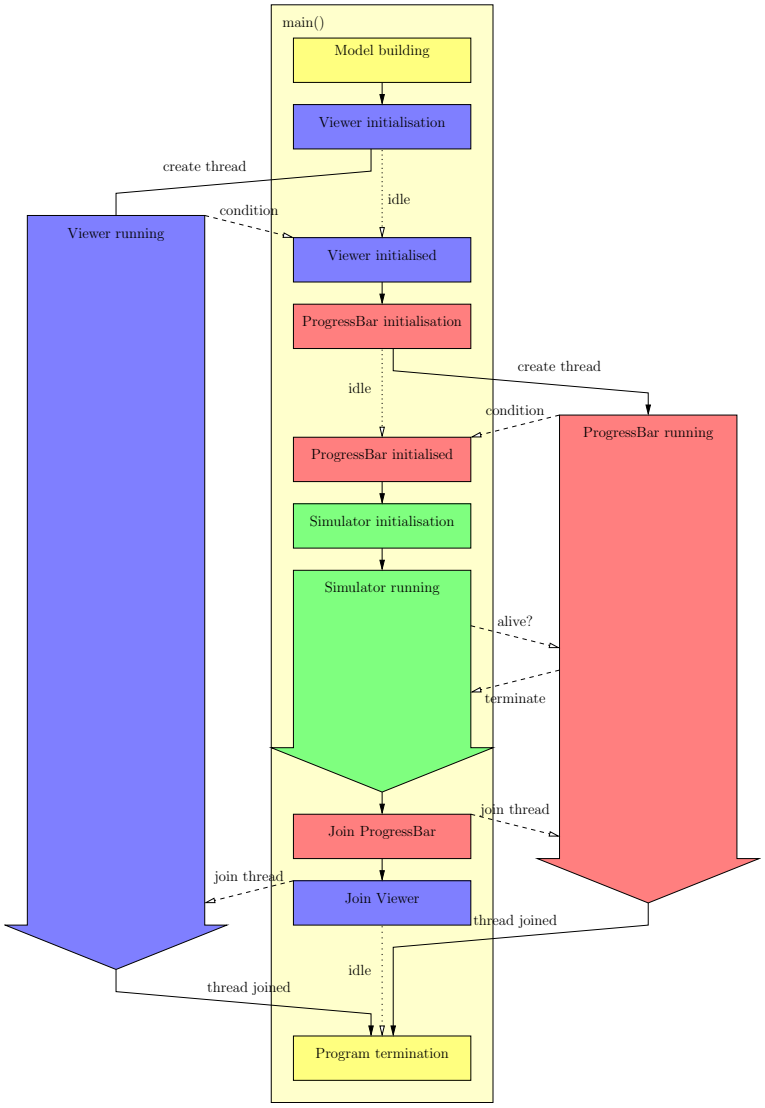


Figure 2.4: Threading in `MesoRD`. Once the internal representation of the model is built, the main program will create a thread for the viewer, unless, of course, the viewer has been disabled. The main program will not continue until the graphics are successfully initialised. Initialisation completion is signalled by setting a condition variable. Similarly, a thread for the progress bar is spawned. Once these threads are running, simulation begins. During simulation, the simulator polls the progress bar at suitable times to determine whether simulation should continue or not. Should the progress bar inform the simulator that the user wishes to end simulation, the simulator will terminate gracefully. The main program will then join the threads for the progress bar and the viewer.

Chapter 3

System Geometry

Volumetric system geometry is essential to the way MesoRD works: simulating diffusion in three dimensions would be difficult without some space to diffuse in. As was mentioned in section 2.3.6, each compartment in MesoRD must have its own geometry definition. The geometry of the entire system is the *union* of all compartment geometries. The volume of the system geometry must be discretized in terms of subvolumes before it is ready for simulation use. Current versions of SBML have no means to define such geometries [33]. However, the issue is being worked on, and in a not too distant future there may be standard ways to deal with geometries. In the meantime, this entire chapter describes a MesoRD-specific extension to SBML.

Irrespective of *how* system geometry is defined, the next subvolume method requires the total volume, V, to be divided into N subvolumes, each with volume Ω. Since subvolumes are inherently three-dimensional, a geometry definition language for MesoRD is not required to handle anything but volumes[1]. The software must be able to keep track of the spatial relationship among the subvolumes: every subvolume must "know" about its neighbouring subvolumes, as the latter are potential targets of diffusion events. Connectivity is discussed in section 3.3.

3.1 Constructive Solid Geometry

Neither users nor developers of MesoRD are expected to be highly skilled mathematicians. Therefore, MesoRD's geometry language should be easy to speak, yet powerful enough to enable construction of non-trivial geometries. A very intuitive way to describe geometry comes from computer aided design and is also used in many three-dimensional modelling packages. It is called *constructive solid geometry*, or CSG for short.

CSG gives a high-level description of geometry. It is intuitive because its syntax resembles the way humans may describe objects in space. Reverting

[1] In MesoRD it is actually impossible to simulate systems with spatial dimensionality different from three. However, surfaces may be modelled by ensuring the thickness of the geometry is much smaller than its area. Similarly, curves will need to be very thin in directions orthogonal to the tangent. Because any physical object in *The Real World*^{TM} extends in all three dimensions, MesoRD's idea of a curve or a surface may arguably be more realistic than a *true* one- or two-dimensional object.

Figure 3.1: A simple two-dimensional object. This could be described as "a circle with four rectangles sticking out its sides" or "two rectangles crossed on top of a circle".

for a moment to two dimensions, the object in Fig. 3.1 could, sacrificing any mathematical accuracy, be described as "a circle with four rectangles sticking out its sides" or "two rectangles crossed on top of a circle". Having a computer come up with such a description is surprisingly hard, mainly because humans, unlike computers, can "see" the circle in spite of the fact that its characteristic circumference is interrupted by rectangles. As it turns out, having the same computer *generate* a geometry from such a description is relatively simple.

Theoretically, CSG is a functional procedure that defines a combined, solid, volumetric object from a number of primitive solid volumes by the application of bounded, Boolean set operations [46, 47]. The set of primitive volumes varies with the particular CSG implementation, while the set of operations is quite similar across the board. `MesoRD`'s set of primitives, the parameters needed to define them as well as transformations applicable to them, are inspired by the Virtual Reality Modelling Language, VRML [48]. Note that standard VRML does *not* actually support CSG – its geometric primitives cannot be combined using set operations. Also, VRML is not an extended markup language. There are efforts underway to address both these issues.

In the spirit of the rest of this document, the current chapter does not go into great mathematical detail. The purpose of the present material is to describe the automatic conversion from a geometry description to a subvolume discretization. Most of the necessary theory was assembled by Requicha and Tilove in 1978 [49].

3.1.1 Tree Representation of CSG

In chapter 4, mathematical expressions consisting of operators and operands will be represented by trees. A CSG procedure can similarly be represented by a tree structure. The root of the tree defines the object of interest, and the leaf nodes are geometric primitives. In between the root and the leaves lie operator nodes. The root object is determined by combining the leaves according to the operator nodes. In a CSG procedure, the operators are either transformations or set operations. A simple, two-dimensional, CSG procedure without transformations is illustrated in Fig. 3.2.

In general, the Boolean-valued set operations used to combine the primitives

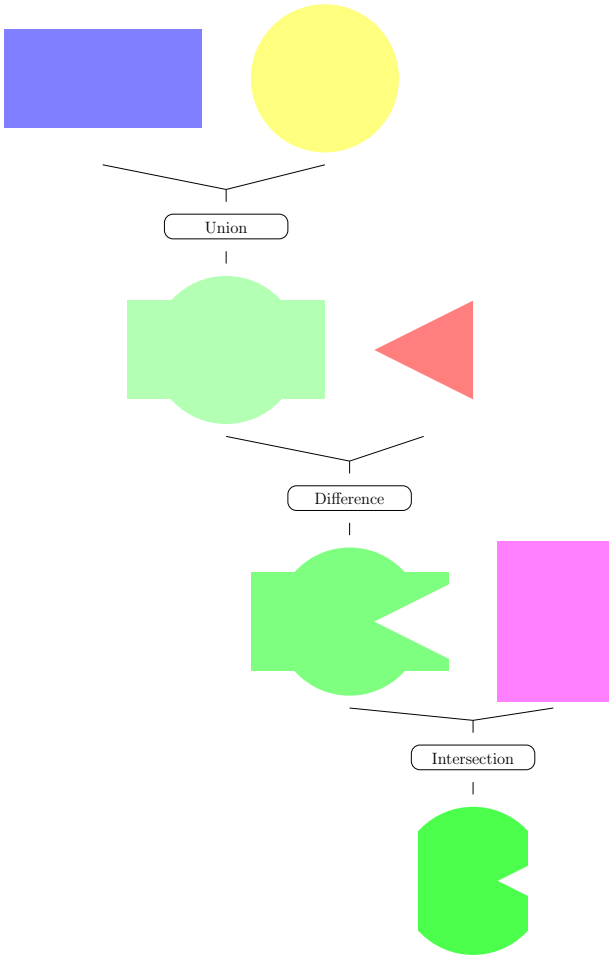


Figure 3.2: Set operations and geometric primitives in a simple two-dimensional CSG procedure. This particular specimen is one of the few trees in computerland that has its root pointing towards the bottom of the page. The blue box is combined with the yellow circle in a union operation, resulting in the upper green object. The red triangle is subtracted from the green object using the difference operator. The last operation, intersection, is illustrated at the bottom part of the figure. For clarity, all transformation operators have been omitted. Note also that this is a binary tree. In general, CSG trees need not be binary: union and intersection nodes can have any number of children.

are not commutative. Therefore, the edges of the tree need to be ordered. It is not difficult to see that exchanging the operands of a difference operation will yield different results, just as $a - b \neq b - a$. The ordering of operands is relevant from a computational aspect as well: if a point in space is contained by the first operand of a union, the algorithm need not spend time checking for containment in the remaining operands.

Other properties of the tree representation of a CSG procedure are:

- The leaf nodes always define a geometric primitive. The inverse is also true: every geometric primitive is represented by a leaf node. By definition, leaf nodes do not have any children.
- Transformation nodes only have one single child.
- Difference nodes have exactly two children, such that the sought object is defined by the volume of the left node *minus* the volume of the right node. Intersection and union nodes can have any number of children.
- A CSG tree does not provide a unique representation. As was exemplified in the figure caption, there are at least two ways to describe the object in Fig. 3.1. Most objects could be described by several different CSG procedures.

In the world of `MesoRD`, every volume element of the full geometry must belong to exactly one compartment. Because every compartment must define its own CSG tree, it is possible to specify systems with overlapping CSG trees, so that some volume element is "owned" by several compartments. This is not an error *per se*, but merely a bad thing to do, as it will cause confusion when it comes to assigning the volume element to a compartment. The scenario is easily avoided using the difference set operation as described in section 3.2.4. The responsibility of avoiding these situations lies entirely on the side of the user.

`MesoRD`'s geometries are defined by a hierarchical, textual, XML representation of the desired CSG procedure. The XML representation is merely a transcription of the CSG procedure in its tree form. Each element in the XML hierarchy is identified as either a set operation node, a transformation node or a primitive geometry node. The parameters needed to completely define each node, such as the side lengths in case of a box, or the displacements in case of a translation transformation, are supplied as attributes. The XML is kept as an annotation of the compartment, the geometry of which the corresponding CSG procedure defines. Further practical details about geometry definition may be found in the User's Guide [34].

3.2 CSG Trees in `MesoRD`

3.2.1 Geometric Primitives and Containment Testing

Primitives in `MesoRD` can be boxes, cones, cylinders, meshes and spheres. Just like objects in VRML, primitives are defined centred on the origin of a *local*, right-handed coordinate system. In the case of cones and cylinders, the axis of symmetry is defined to lie parallel to the local y-axis. The local coordinate

systems are related to the *global* coordinate system, the right-handed coordinate system of the full geometry, by a set of affine transformations.

The most important operation on primitives is containment testing: given a point $\mathbf{p} = [x_p,\ y_p,\ z_p]^{\mathrm{T}}$ in the local coordinate system, `MesoRD` needs to figure out whether $\mathbf{p}$ is contained in the primitive or not. The naive algorithms work well for all primitives except for the mesh.

The Triangular Mesh

There is no CSG "standard". Meshes are "non-standard" only in the sense that many CSG implementations do not support them. In `MesoRD`, containment testing for meshes is non-standard, too.

For triangular meshes, three two-dimensional arrays for the xy-, xz- and yz-planes respectively are generated. Each array element holds a set of ranges. These ranges indicate the regions of a ray, originating at the coordinates corresponding to the point in the array and perpendicular to the plane represented by the array, that lie inside the mesh. Ranges are determined by checking the intersection of the ray with the set of triangles that comprise the mesh. A ray that hits the front of a triangle marks the start of a range. When the same ray hits the back of a triangle the range is ended. For instance, the xy-array at $(x_p,\ y_p)$ contains a range set $\{[z_{f_i},\ z_{b_i}]\}_{i=0}^{n-1}$ of cardinality n. If $z_p \in [z_{f_i},\ z_{b_i}]$ for any $0 \leq i \leq n-1$, $\mathbf{p}$ is contained in the mesh.

One may think that keeping one array would be sufficient. It probably would, if it was not for numerical inaccuracies. When a ray strikes a triangle tangent to the triangle plane, it is not easy to tell whether it has hit the front or the back of the plane. Therefore, range determination in all directions is done conservatively, ignoring any ranges arising from intersection angles close to $\pi/2$ with the triangle normal. If $\mathbf{p}$ is in the range for any of the xy-, xz- or yz-arrays, it is contained in the mesh.

The mesh is probably the most powerful primitive available in `MesoRD`'s CSG implementation, as it allows for the creation of natural shapes. However, since an object is composed of many triangles, which in turn are defined by three vertices, very many XML elements are required to describe even simple shapes. From a design perspective this is permissible, as SBML was never intended to be hand-written.

3.2.2 Set Operations

The set of transformed primitives, or objects, are combined using difference, intersection and union set operations. These operations take two or more operand objects and combine them into a new object. The unary complement operation, which in combination with the intersection operation would obsolete the difference operation, is not supported, since `MesoRD` does not define a *universe*.

Objects resulting from a set operation can act as operand objects in other set operations. The necessary proofs, albeit under the assumption of a well-defined universe[2] have been collected by Requicha and Tilove [49].

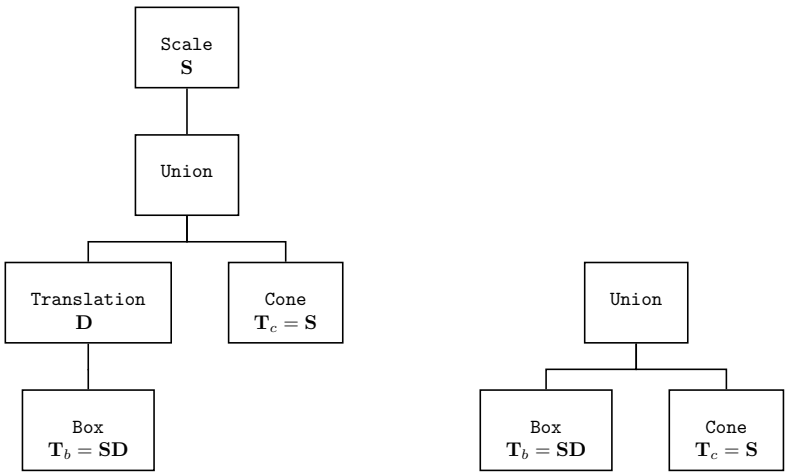
[2]For the purpose of Requicha and Tilove's theory, one may define the universe in `MesoRD` CSG as the *bounding box* of the geometry, see section 3.5.

3.2.3 Transformations

Clearly the shape of a compound object depends on the exact location and orientation of the individual primitives. Therefore, CSG trees in `MesoRD` contain transformation nodes in addition to the set operation nodes. Transformations allow subtrees to be rotated, scaled, sheared and translated. These are all affine transformations, preserving the parallelism of lines, but not lengths or angles. Every primitive stores its own transformation matrix which relates its local coordinate system to the global coordinate system. For example, a cylinder parallel to the x-axis is defined by rotating a cylinder node by $\pi/2$ around the z-axis. In case no transformation is specified, the unit transformation is assumed. Transformations have no effect on set operations, but are propagated to its children.

A transformation node will affect all leaf nodes below it in the tree. The transformation that will take a primitive leaf node from its local coordinate system to the global coordinate system is defined by the composition of all transformations on the path between the root and the respective leaf. The compound transformation is calculated as the matrix product of the corresponding transformation matrices.

An example which describes the union of a cone and a box is illustrated in Fig. 3.3. The root of the tree is a scale node, which defines a scale matrix $\mathbf{S}$. The path from the root to the cone primitive only passes through one transformation node, namely the root. Thus, the transformation matrix relating the local coordinate system, in which the cone is defined, to the global coordinate system of the entire procedure, is $\mathbf{T}_c = \mathbf{S}$.



(a) The original CSG tree.

(b) The pruned CSG tree.

Figure 3.3: A sample CSG tree as it would have been represented in the `MesoRD` implementation. The tree describes the union of a cone and a box. The box is translated with respect to the cone. The union of both primitives is scaled. A pruned version of the tree in Fig. 3.3(a) is shown in Fig. 3.3(b).

The box node is separated from the union node by a translation node which defines a displacement matrix $\mathbf{D}$. The path from the box primitive to the root passes through two transformation nodes. The transformation matrix that relates the local coordinate system of the box to the global coordinate system is $\mathbf{T}_b = \mathbf{SD}$.

Once the compound transformation $\mathbf{T}_i$ has been computed and stored for every leaf i, the transformation nodes can be pruned without loss of information. In `MesoRD` pruning is done while the internal representation is built from the XML definition – at no point does the internal `MesoRD` CSG tree contain any transformation nodes.

Homogeneous Coordinates

Throughout the CSG procedure, `MesoRD` internally represents points in space by four-dimensional homogeneous coordinates [43, 47]. A vector in the usual, three-dimensional, Cartesian space is converted into a homogeneous, four-dimensional ditto by appending an element with value $w = 1$. The reverse operation, homogenisation, is illustrated in Eq. 3.1.

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \mapsto \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{bmatrix} \tag{3.1}$$

In other words, one defines the usual, three-dimensional subspace by the hyperplane $w = 1$ through homogeneous space. From the transformation in Eq. 3.1, one can see that two points in homogeneous coordinates map to the same point in Cartesian space if one is a non-zero multiple of the other.

A 4×4 homogeneous transformation matrix is constructed similar to the four-dimensional homogeneous vectors. The desired result is obtained by starting from a four-dimensional identity matrix and replacing the top, leftmost 3×3 sub-matrix by the usual three-dimensional transformation matrix. The reverse operation is analogous, although, as will be seen, not always possible.

Using homogeneous coordinates, compound transformations involving translations are multiplicative instead of additive. For instance,

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

defines a translation by d_x, d_y and d_z in the x-, y- and z-directions respectively. Since the top elements of the fourth column are non-zero, it is not possible to homogenise $\mathbf{D}$ to the usual, three-dimensional representation. In fact, it is impossible to express translation as a 3×3 multiplicative transformation matrix. This is the reason why `MesoRD` uses four-dimensional homogeneous, instead of three-dimensional Cartesian, coordinates.

3.2.4 The `compartment` Primitive

`MesoRD` supports one more primitive that was not mentioned in section 3.2.1. The `compartment` primitive is a special pseudo-primitive that allows the ge-

ometry of a previously defined compartment to be treated just like any other geometric primitive.

With the `compartment` primitive it is possible to make use of a previously defined compartment in the definition of another. While this "copy operation" allows for new creative ways in the definition of geometries, the main reason for its inception is to allow one compartment to surround another. As was mentioned in section 3.1.1, it is not a good idea to define geometries so that a point in space could belong to several compartments. Using the `compartment` primitive together with the difference set operation, the *containing* compartment may be defined as the left operand of a difference operation. The right operand is set to the geometry of the *contained* compartment. This will create a hole in the containing compartment, just big enough to fit the contained compartment.

3.3 Connectivity

In order to establish connectivity among subvolumes, `MesoRD` will need to find which subvolumes are related by a neighbour-relationship. In particular, given a subvolume $\Omega_{\mathbf{p}}$, centred at $\mathbf{p}$, and a directional vector $\mathbf{d}$, `MesoRD` must be able to find a reference to $\Omega_{\mathbf{p}}$'s neighbour in direction $\mathbf{d}$. If $\mathbf{d}$ is scaled such that its length is equal to the size of a subvolume, ℓ, the neighbour in direction $\mathbf{d}$, $\Omega_{\mathbf{p_d}}$, will be centred at $\mathbf{p_d} = \mathbf{p} + \mathbf{d}$. In case no subvolume exists at $\mathbf{p_d}$, $\Omega_{\mathbf{p}}$ is a boundary subvolume in direction $\mathbf{d}$. The algorithm outlined here would be sufficient, were it not for periodic boundary conditions.

3.3.1 Periodic Boundary Conditions

Unlike many other CSG implementations, `MesoRD` supports periodic boundary conditions, PBC:s, for boxes and cylinders. For a box, PBC:s mean that a pair of opposing sides are topologically equivalent: if a molecule diffuses out one side, it will reappear at the other. For a cylinder[3], periodic boundary conditions, or toroidal boundaries, mean that molecules leaving through one circular cap will reappear at the other. Periodic boundary conditions have an important history in the sort of simulations `MesoRD` deals with, since they allow for the creation of "infinite" geometries [12].

With periodic boundary conditions, the algorithm first determines whether $\Omega_{\mathbf{p}}$ lies on the boundary of the geometric primitive that includes it. Being a boundary subvolume of the including primitive does not mean it is a boundary subvolume of the *total* geometry: if the geometry in Fig. 3.1 is defined by the union of two crossed boxes and a sphere, the boxes have *virtual boundaries* inside the sphere. These boundaries are not *true* boundaries of the total geometry, since they are contained within the sphere.

Periodic boundary conditions on virtual boundaries are pathological cases: it makes little sense to define an object with PBC, only to subsequently hide the periodic boundaries within a bigger volume. Such situations should generally be avoided; periodic boundaries should also be true boundaries.

[3]Periodic boundary conditions are the reason why `MesoRD` uses separate primitives for cones and cylinders. Cones could more generally be described by their height, h, the radius of the bottom cap, r_b, and the radius of the top cap, r_t. A "proper" cone would have $r_t = 0$, and a cylinder would be defined by the special case $r_b = r_t$. Since PBC:s require topologically equivalent sides to be *exactly* the same size, they would not be applicable to generalised cones.

Should the subvolume, $\Omega_{\mathbf{p}}$, be a boundary of the defining primitive in direction $\mathbf{d}_{PBC}$ *and* periodic boundary conditions are active in direction $\mathbf{d}_{PBC}$, where $\mathbf{d}_{PBC} \cdot \mathbf{d} \neq 0$, `MesoRD` attempts to find a reference to the subvolume at $\mathbf{p}_{PBC} = \mathbf{p} + \mathbf{d} - l_{PBC}\mathbf{d}_{PBC}$ instead. Here, l_{PBC} is the length of the geometric primitive in direction $\mathbf{d}_{PBC}$, given in units of subvolume size, ℓ. For example, if $\mathbf{d} = \mathbf{d}_{PBC}$ points one subvolume to the right, $\mathbf{d} - l_{PBC}\mathbf{d}_{PBC}$ points $l_{PBC} - 1$ subvolumes to the left. Because the primitive is l_{PBC} subvolumes wide in direction $\mathbf{d}_{PBC}$, $\mathbf{p} + \mathbf{d} - l_{PBC}\mathbf{d}_{PBC}$ points to the subvolume opposite of $\Omega_{\mathbf{p}}$, providing $\Omega_{\mathbf{p}}$ is a right boundary subvolume.

In the `MesoRD` implementation the `getNeighbour()` function will, if possible, retrieve a reference to the neighbour in the sought direction.

3.4 `MesoRD` CSG Algorithm

The CSG implementation of `MesoRD` proceeds in two passes. A second pass is needed because periodic boundary conditions imply that subvolumes that are not spatially close can nevertheless be neighbours. Thus, neighbour-resolution under PBC may require retrieval of references to subvolumes which have not yet been built. A pseudo-code version of the algorithm is given in Alg. 8.

First of all, the bounding box for every compartment geometry is computed. The bounding box, or extent, of a CSG procedure is defined as the smallest box, the sides of which are parallel to the axes of the coordinate system, that completely contains the result of the procedure. Bounding boxes are discussed in more detail in section 3.5. From the compartment bounding boxes, the *global* bounding box, which is bounding box that contains the union of all compartment geometries, is constructed. The space within the global bounding box is treated as a point lattice, so that every lattice point is a potential centre of a subvolume.

- In the first pass, the lattice within the global bounding box is traversed in a set order. By testing whether the CSG procedure for a compartment contains the lattice point, the algorithm decides whether a subvolume, centred at the point, should be generated or not. For each compartment, c, a list, $\mathcal{L}_c$ is constructed which holds the subvolumes belonging to that particular compartment. If a subvolume lies on a periodic boundary, it is added to another list, $\mathcal{B}$, as well.

- In the second pass, every subvolume in $\mathcal{B}$ is inspected. The neighbour-relationships in all directions are updated using the procedure described in section 3.3.1.

The non-PBC neighbour-relationships can be resolved while generating subvolumes in the first pass. If space is traversed in order of increasing x, y, and z, the neighbouring subvolumes in the negative x-, y-, and z-directions are always known. For example, when adding a new cubic subvolume at $[x,\ y,\ z]^{\mathrm{T}}$ three of its neighbours can be immediately assigned. One merely needs to query $\{\mathcal{L}_c\}$ for the subvolumes centred at $[x-1,\ y,\ z]^{\mathrm{T}}$, $[x,\ y-1,\ z]^{\mathrm{T}}$ and $[x,\ y,\ z-1]^{\mathrm{T}}$. Doing so, one must not forget to update these neighbouring subvolumes as well, so that they are aware of the recently added subvolume. A newly constructed subvolume is always treated as a non-periodic boundary subvolume in the positive x-, y-, and z-directions.

Algorithm 8: The CSG algorithm of `MesoRD`. The input to the algorithm is a set of n_c CSG trees, $\{t_c\}_{c=0}^{n_c-1}$, so that the each tree describes the CSG procedure for exactly one compartment. The algorithm will produce a set of lists, $\{\mathcal{L}_c\}_{c=0}^{n_c-1}$, of the subvolumes in each compartment.

```
begin
    G ← global bounding box computed from {t_c}
    foreach p ∈ G do
        for c ← 0 to n_c − 1 do
            if contains(t_c, p) ≠ exterior point then
                Ω_new ← new subvolume centred at p
                L_c ← L_c ⋃ Ω_new
                if contains(t_c, p) = boundary point then
                    B ← B ⋃ Ω_new
                Resolve non-PBC neighbour relationships
    foreach Ω ∈ B do
        Resolve PBC neighbour relationships
end
```

3.5 Bounding Boxes

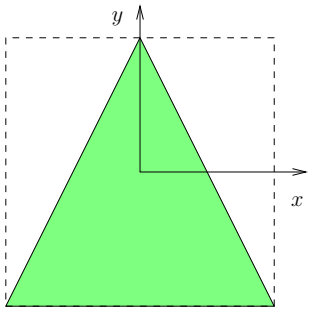
Because the bounding boxes are the smallest boxes that completely contain the result of the CSG procedure, they can be used as first approximations to the final geometry. Because everything of interest must be contained inside the global bounding box, one could imagine the global bounding box to also limit the *universe* of the CSG procedure.

Because the local coordinate system is related to the global coordinate system by a compound transformation matrix $\mathbf{T}_i$, the bounding box in the local coordinate system will transform as $\mathbf{T}_i$ into the global coordinate system. The local bounding box always has sides parallel to the planes of the local coordinate system, but as soon as $\mathbf{T}_i$ involves any rotation or shear, the transformed bounding box will not be similarly aligned to the planes of the global coordinate system. Due to this misalignment and the definition of a bounding box, the transformed local bounding box is *not* a proper bounding box in the global coordinate system. This is illustrated in Fig. 3.4.

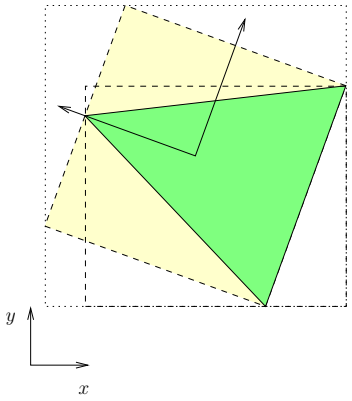
There are reasons for minimising the size of the bounding box, apart from satisfying the definition of a bounding box. As may have become apparent while `MesoRD`'s CSG algorithm was described in section 3.4, any sloppiness in bounding box computation will incur a decrease in performance due to unnecessary traversal of empty space.

In order to construct the global bounding box of a transformed primitive, `MesoRD` transforms the extreme points of the convex hull of the primitive, and deduces the bounding box from the transformed extreme points. The box is the simplest example: its global bounding box is the smallest box that contains the eight transformed corners of the original box.

The quadrics are a little trickier. To find the bounding box of, for instance, a cylinder, `MesoRD` transforms a *parametrisation* of the circular base and another similar parametrisation of the circular cap. The x- and y-coordinates



(a) The local bounding box.



(b) The global bounding box. The local bounding box is indicated by the shaded rectangle. The bounding box of the transformed local bounding box is shown by the dotted rectangle.

Figure 3.4: The bounding box, or rather, the "bounding rectangle", of a simple two-dimensional object. The dashed lines describe the smallest rectangle, the sides of which are aligned with the axes of the coordinate system, that completely contains the object. The bounding box in the local coordinate system is shown in Fig. 3.4(a). The bounding box in the global coordinate system, which is related to the local coordinate system by some transformation $\mathbf{T}_i$, is shown in Fig. 3.4(b). Note that the global bounding box is *not* the bounding box of the transformed local bounding box, since that would no longer be the *smallest* box that completely contains the object.

of an untransformed cylinder can be parametrised in terms of $\varphi \in [0,\, 2\pi)$ to $[r \cdot \cos(\varphi)\,,\; r \cdot \sin(\varphi)]^{\mathrm{T}}$, where r is the cylinder's radius. Its circular base lies at $z = -h/2$. Transformation by $\mathbf{T}_i$ yields

$$\mathbf{T}_i\mathbf{c}_b = \begin{bmatrix} t_{00} & t_{01} & t_{02} & t_{03} \\ t_{10} & t_{11} & t_{12} & t_{13} \\ t_{20} & t_{21} & t_{22} & t_{23} \\ t_{30} & t_{31} & t_{32} & t_{33} \end{bmatrix} \begin{bmatrix} r \cdot \cos(\varphi) \\ r \cdot \sin(\varphi) \\ h/2 \\ 1 \end{bmatrix}. \tag{3.2}$$

In order to find the extrema of the transformed base, the vector resulting from Eq. 3.2 is differentiated with respect to φ

$$\frac{\partial \mathbf{T}_i\mathbf{c}_b}{\partial \varphi} = r \cdot \begin{bmatrix} t_{01} \cdot \cos(\varphi) - t_{00} \cdot \sin(\varphi) \\ t_{11} \cdot \cos(\varphi) - t_{10} \cdot \sin(\varphi) \\ t_{21} \cdot \cos(\varphi) - t_{20} \cdot \sin(\varphi) \\ t_{31} \cdot \cos(\varphi) - t_{30} \cdot \sin(\varphi) \end{bmatrix}. \tag{3.3}$$

Finding the extrema of the transformed circular base now amounts to back-substituting boundary points of $\mathbf{T}_i\mathbf{c}_b$, as well as the φ where any component of $\partial \mathbf{T}_i\mathbf{c}_b/\partial\varphi$ is zero, into Eq. 3.2. The latter φ must satisfy

$$\begin{cases} \varphi_0 = \arctan(t_{01}/t_{00}) \\ \varphi_1 = \arctan(t_{11}/t_{10}) \\ \varphi_2 = \arctan(t_{21}/t_{20}) \\ \varphi_3 = \arctan(t_{31}/t_{30}) \end{cases}. \tag{3.4}$$

Spheres an cones are handled similarly.

In the `MesoRD` implementation the `getBoundingBox()` function is used to retrieve the bounding box of a CSG procedure.

3.5.1 Bounding Boxes of Set Operations

Since transformation nodes are pruned early, the CSG procedure inside `MesoRD` only contains set operation and primitive nodes. Because the algorithm described in this chapter will need to compute the bounding box for the *entire* CSG procedure, a way to calculate bounding boxes for set operations is needed.

- The bounding box of a difference operation is the bounding box of the left operand. Not even when the bounding box of the left operand completely contains the bounding box of the right operand is it safe to assume that the resulting bounding box is anything less than the bounding box of the left operand. An example illustrating this is shown in Fig. 3.5.
- The bounding box of an intersection is the intersection of the bounding boxes for all operand nodes in the operation.
- The bounding box of a union is the union of the bounding boxes for all nodes in the operation.

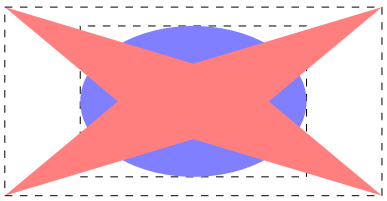


Figure 3.5: The bounding boxes, indicated by dashed lines, in a difference operation. The red object is to be subtracted from the blue object. The bounding box of the red object completely contains the bounding box of the blue object. However, the bounding box of the resulting object will still be the bounding box of the blue object.

Chapter 4

Expression Evaluation

4.1 Reaction Rates

As was seen in chapter 1, reaction rates and reaction probabilities are, at least from a computational viewpoint, equivalent. Consider for instance the reaction

$$S_i + S_l \xrightarrow{k/q} S_j + S_l. \tag{4.1}$$

Assuming that the reaction in Eq. 4.1 is a second order reaction, catalysed by S_l, the *rate* of the reaction, or its probability of occurrence per time unit, can be written

$$a = \frac{{}^{\#}s_i \cdot \ {}^{\#}s_l \cdot k}{q},$$

just as was done in section 1.3.

Rates in SBML are mathematical expressions, defined in a subset of MathML [33]. These expressions may depend on the current composition of the subvolume where the reaction is taking place, present parameter values and external functions, defined elsewhere in the model. SBML requires all rate laws to evaluate to s^{-1}. In the above example, a must thus have the unit s^{-1}, and because ${}^{\#}s_i$ as well as ${}^{\#}s_l$ are dimensionless quantities, k/q must also have the unit s^{-1}. The topic of dimension analysis is further discussed in section 4.3.

Let the instantaneous composition of a subvolume be defined by a state vector $\mathbf{x}$, as first encountered in section 1.3. The number of molecules of the i:th species is given by the i:th element of $\mathbf{x}$, x_i. Similarly, assume that parameter values are stored in a vector $\mathbf{p}$. Let the index in $\mathbf{p}$ for any parameter be given by a function `pIndex(`*identifier*`)`. Thus, the rate of the reaction in Eq. 4.1, can be written as a function:

$$a = a\left(\mathbf{x}, \mathbf{p}\right) = \frac{x_i \cdot x_l \cdot p_{\texttt{pIndex}(k)}}{p_{\texttt{pIndex}(q)}}. \tag{4.2}$$

The MathML corresponding to Eq. 4.2 is shown in Fig. 4.1.

Currently, `MesoRD` has no support for user-defined functions. Once `MesoRD` implements functions, rate expressions will depend on a vector of functions, $\mathbf{f}$, as well.

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <apply>
    <divide/>
    <apply>
      <times/>
      <ci>Si<ci/>
      <ci>Sl<ci/>
      <ci>k<ci/>
    </apply>
    <ci>q<ci/>
  </apply>
</math>
```

Figure 4.1: The rate as defined by Eq. 4.2 expressed in MathML. $^{\#}s_i$ is assumed to be defined by the species identified by `Si`. $^{\#}s_l$ is similarly related to `Sl`. Note that SBML only supports a subset of MathML.

4.2 The Abstract Syntax Tree

MathML, like SBML, is an XML. The inherently hierarchical structure of extended markup languages means that it is natural to interpret MathML as a tree. In this context, the trees are called *abstract syntax trees*, or AST:s. Building the AST from the information given in MathML/SBML is easy; most of the work is done by the XML engine in cooperation with `libsbml` [32, 35, 36]. For instance, the expression from Eq. 4.2 has the AST shown in Fig. 4.2.

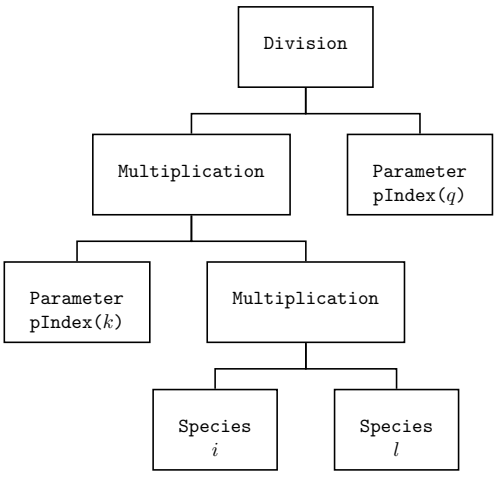


Figure 4.2: The abstract syntax tree, or AST, corresponding to Eq. 4.2. This is a tree representation of the information in Fig. 4.1. Nodes should only store indices of species and parameters into their respective vectors, not the objects themselves.

4.2.1 AST Evaluation

The AST is the starting point for all procedures dealing with reaction rate evaluation. Evaluating an AST means finding the value of its root node. This value can be found by recursive depth-first traversal, evaluating every node as it is encountered. Evaluation thus requires $\mathcal{O}(n)$ operations, where n is the number of nodes in the tree.

Reaction rate evaluation is done frequently; as soon as the composition of a subvolume is changed, the rates for all reactions that depend on the changed species must be recomputed. Reaction rate evaluation could easily become a performance bottleneck, and care should be taken to make its implementation as efficient as possible. An attempt at improving the performance of repeated, identical, rate evaluation was discussed in section 2.3.5.

To further improve performance, `MesoRD` attempts to transform the tree, so that it becomes faster to evaluate. Because performance scales linearly with the number of nodes, a way to accomplish node reduction is examined in section 4.4. An experimental feature in `MesoRD` is a transformation that will convert certain AST:s to a linear form. The linearised AST can subsequently be evaluated iteratively. Linearisation is discussed in section 4.5.

In order to resolve the parameter identifiers and species indices[1], present in an AST, the evaluator needs to access the symbol tables discussed in chapter 2. Such look-ups can be expensive. Unfortunately, there is no alternative. Only if parameters, as well as species compositions, were constant, could look-ups be completely avoided. But then, there would also be no need for rate evaluation: if a subvolume's chemical composition does not change, the probabilities of the reactions will not change either. A system in which everything is constant is probably too boring to simulate.

4.3 Dimension Analysis

Once the AST has been constructed, it is possible, using functions from the `Unit` class, to compute the unit of the expression. The unit of the expression can be found by a depth-first traversal, recursively determining the unit of every node, similar to how one would evaluate the AST. Only if the model defines units for all parameters used in an expression, is it possible to assign a unit to the root of the corresponding tree. Including units in parameter definitions is optional in SBML. Thus, dimension analysis is not always possible.

As was mentioned in section 4.1, rate units should evaluate to s^{-1}. This is a little awkward for the type of simulations `MesoRD` deals with, because doing so would require one to include the reaction volume in the rate expression[2]. The reaction volume is identical to the volume of a subvolume, and this quantity, in turn, is not necessarily a part of the model description.

`MesoRD` allows reactions to be specified in $\mathrm{molar} \cdot \mathrm{s}^{-1} = \mathrm{mole} \cdot \mathrm{l}^{-1} \cdot \mathrm{s}^{-1}$. If `MesoRD` finds that this is the case, it will multiply the rate expression by

[1] Only indices of species and parameters into their respective species and parameter vectors should be stored in the tree. This is because up-to-date values of all involved, non-constant, quantities must be retrieved while evaluating an AST. While the AST itself is static, the symbol tables always hold the current values of any parameters or species.

[2] As was noted in section 1.3.1, the microscopic reaction rates, unlike the macroscopic dittos, are volume-dependant.

the volume of a subvolume and the reciprocal of Avogadro's number. Since the volume of a subvolume is constant within the simulation, the resulting expression may be factored as described in section 4.4 below. This transformation will ensure that the rate has the unit s^{-1}, and the user does not need to include the volume of a subvolume in the model definition. If the rate expression already evaluates to s^{-1}, or if the unit cannot be determined, `MesoRD` will assume the user knows what the user does, and proceed without altering the expression at all. Thus, `MesoRD` has relaxed the SBML specification a little, not restricted it.

4.4 Constant Folding

Parameters in SBML can be defined to be constant, and `MesoRD` can make use of this information. The value of a constant parameter can be looked up once and for all while constructing the AST. `MesoRD` will then replace the `Parameter` node with a `Constant` node, the value of which is the value of the parameter. During simulation, the evaluator need not spend any time consulting the `ParameterTable`.

Assume that the parameters k and q in Eq. 4.2 are constant. This means the expression can be simplified.

$$a = x_i \cdot x_l \cdot m \text{ where } m = \frac{p_{\texttt{pIndex}(k)}}{p_{\texttt{pIndex}(q)}} \text{ is constant} \tag{4.3}$$

Redrawing the AST, one would see that the number of nodes in the AST has decreased by two. Since evaluation scales linearly with the number of nodes in the tree, this is an improvement.

Common techniques from compiler design can be used to implement improvements such as described above [17]. `MesoRD`'s constant folding and algebraic simplification functions will scan the tree for patterns which can be simplified. For instance, if both children of a multiplication node are constant, the entire subtree can be replaced by a new `Constant` node, the value of which is the product of the two multiplication operands. The operation is illustrated in Fig. 4.3.

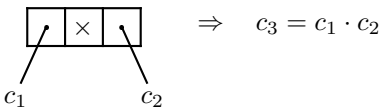


Figure 4.3: Algebraic simplification of a multiplication operation with two constant operands. The left and right operands are denoted c_1 and c_2 respectively. The subtree on the left is replaced by a new constant node, the value of which, c_3, is the product of the two constant operands.

The process can be repeated on a full tree until no more transformations can be applied. All transformations implemented in `MesoRD` are given in appendix A.

4.5 Expression Linearisation

MathML is powerful, and even the subset implemented in SBML is quite expressive. However, it is likely that most `MesoRD` simulations will be subject to rate expressions that are reasonably simple. A typical rate expression is assumed to have the form

$$a = \sum_{i=0}^{m} \left(\alpha_i \prod_{j=0}^{n} f_{ij} \right) \text{ where } f_{ij} = \left({}^{\#}s_{ij} + \beta_{ij} \right)^{\gamma_{ij}}, \tag{4.4}$$

where, α, β and γ are constants. Once constant folding has been applied to the AST, `MesoRD` will attempt to express the AST on the form of Eq. 4.4. If this succeeds, reaction evaluation can be implemented as an iterative procedure. Because m and n in Eq. 4.4 are typically small integers[3], loop unrolling is feasible.

[3]Since most chemical reactions do not involve more than two species, $m \leq 2$ and $n \leq 2$ is expected to hold in typical cases. More complicated reactions are believed to occur in a sequence of steps, where each elementary step involves only one or two reacting species [7, 10].

Chapter 5

Conclusion

The implementation discussed in this document is licensed under the GNU General Public License. It is available in source and binary form on the world wide web.

`http://mesord.sourceforge.net`

Please use reference [50] when citing `MesoRD` and reference [2] when citing the next subvolume method. The author welcomes comments and suggestions, regarding the implementation as well as this text.

`hattne@users.sourceforge.net`

`MesoRD` was first released in September 2004, and a minor update was posted in January 2005. Deterministic simulation was introduced by David Fange in September 2005, with an update following in April 2006 [51]. The user base is estimated to around 150, which, given the specificity of the problem the software addresses, is not too bad.

The `MesoRD` core seems to work quite well, at least for those involved in its development. On a modern desktop workstation one can simulate systems of around one million subvolumes.

5.1 Related Work

There are several other efforts which are similar in spirit to `MesoRD`. A few of these were summarised by Lemerle *et al.* in 2005 [52]. Comparison and validation of different software simulators, especially with respect to disagreement with mean-field theory, is greatly aided by the recent emergence of public model repositories, such as the BioModels Database [53]. Unlike many other packages, `MesoRD` does not have its own extensive test suit yet.

5.2 Future Work

`MesoRD` is not ready for "production use".

- The most prohibitive deficiency is perhaps its lack of error reporting: when[1] `MesoRD` crashes, it is very difficult to determine why, unless one is equipped with a debugger and an understanding of C++.
- A related shortcoming is the semi-unstructured simulation output. It may be desirable to devise a more widely supported format, better suited for automatic parsing. Output based on XML may be possible, even though its verbosity may cause the already voluminous output to grow even more.
- While `MesoRD` has been tested for basic functionality, it has not been subjected to any performance evaluation. Conducting a profiling analysis may reveal hitherto unidentified bottlenecks, and may also help in answering questions of how efficient intended "improvements" really are.
- There are many new features that could be included in future versions of `MesoRD`. Full SBML support is a prime candidate. By enabling arbitrary growth laws, SBML functions could lead to the exciting possibility of growing and shrinking geometries. One application of variable geometries is simulation of cell division. Dynamic geometries are implemented in `STOCKS` [54]. Similar "shape-distorting processes" were originally planned for `SmartCell` [55].
- While trivial parallelism on the Monte Carlo level is a not an issue[2], multithreaded parallelism of the simulation algorithm definitely is. Such parallelism could perhaps be implemented based on the observation that events and heap reordering need not read or write the same data.
- A parallel implementation for distributed memory architectures is more distant. The connectivity among subvolumes means that extensive communication would be required when the simulation workload is split to several processors. This makes an efficient non-shared memory implementation difficult. The work by Korniss *et al.* may provide a good starting point for working around these problems [56].

Other ideas for improvement are listed in the User's Guide [34].

5.3 Acknowledgements

I wish to thank Johan Elf for giving me the opportunity to work on this tremendously exciting project. Per Lötstedt provided additional supervision and his careful examination of the written report was much appreciated. I thank David Fange for nursing and significantly enhancing the code base, and extending the `MesoRD` development team to a total of three. The project was funded by grants to Måns Ehrenberg from the Swedish Research Council.

Several improvements were suggested by Martin Lovmar and Jesper Gantelius. I thank Paul Sjöberg for testing the first `Solaris` port. Dominic Tölle

[1] Yes, *when* `MesoRD` crashes, not *if*. It turns out to be difficult to write a complete model definition without including errors. On a related note, it turned out to be difficult to write `MesoRD` without including bugs. While model building should ideally be an automated procedure, C++ development probably should not.

[2] This is what Gillespie originally intended to use his algorithm for in 1976 [4].

and Katherine Lawler gave helpful comments on an early draft of this text. Malin Lindgren[3] again played a key role in "quality assurance".

`MesoRD` uses code from several other open-source projects. The Mersenne Twister implementation used by `MesoRD` was adapted by Agner Fog. Parts of the annotation processing are contributed by the Apache Software Foundation. Some tricks on in/out streams are due to Dietmar Kuehl. The matrix implementation was originally written for the `libsim` project. In fact, this entire project would have been a whole lot more difficult without support from the open source community in general.

Finally, a bagful of gratitude goes to the patient users of `MesoRD`, both present and future.

[3]Who, during the course of this project, became Malin Hattne.

Bibliography

[1] Johan Elf, Andreas Dončić, and Måns Ehrenberg. Mesoscopic reaction–diffusion in intracellular signalling. *Proceedings of SPIE*, 5110:114–124, 2003.

[2] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems Biology*, 1(2):230–236, December 2004.

[3] Frank Jensen. *Introduction to Computational Chemistry.* John Wiley & Sons, first edition, 1999.

[4] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22:403–434, 1976.

[5] Larry Lok and Roger Brent. Automatic generation of cellular reaction networks with Moleculizer 1.0. *Nature Biotechnology*, 23(1):131–136, January 2005.

[6] Daniel C. Mattis and M. Lawrence Glasser. The uses of quantum field theory in diffusion-limited reactions. *Reviews of Modern Physics*, 70(3):979–1001, July 1998.

[7] Donald A. McQuarrie. Stochastic approach to chemical kinetics. *Journal of Applied Probability*, 4(3):413–478, December 1967.

[8] Johan Elf. *Intracellular Flows and Fluctuations.* PhD thesis, Uppsala University, 2004.

[9] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[10] P. W. Atkins. *Physical Chemistry.* Oxford University Press, sixth edition, 1998.

[11] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[12] P. Hanusse and A. Blanché. A Monte Carlo method for large reaction–diffusion systems. *Journal of Chemical Physics*, 74(11):6148–6153, June 1981.

[13] Don L. Bunker, Bruce Garret, Tadeusz Kleindienst, and George Stevenson Long III. Discrete simulation methods in combustion kinetics. *Combustion and Flame*, 23:373–379, 1974.

[14] A. B. Bortz, M. H. Kalos, and J. L. Lebowitz. A new algorithm for Monte Carlo simulation of Ising spin systems. *Journal of Computational Physics*, 17(1):10–18, January 1975.

[15] Lennart Råde and Bertil Westergren. *Mathematics Handbook for Science and Engineering*. Studentlitteratur, third edition, 1995.

[16] Jeroen S. van Zon and Pieter Rein ten Wolde. Simulating biochemical networks at the particle level and in time and space: Green's function reaction dynamics. *Physical Review Letters*, 94(128103):1–4, April 2005.

[17] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, first edition, 1997.

[18] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, first edition, 1990.

[19] Steven S. Andrews and Dennis Bray. Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Physical Biology*, 1:137–151, 2004.

[20] A. B. Bortz, M. H. Kalos, J. L. Lebowitz, and M. A. Zendejas. Time evolution of a quenched binary alloy: Computer simulation of a two-dimensional model system. *Physical Review B*, 10(2):535–541, July 1974.

[21] M. Malek-Mansour and J. Houard. A new approximation scheme for the study of fluctuations in nonuniform nonequilibrium systems. *Physics Letters*, 70A(5, 6):366–368, April 1979.

[22] Audrius B. Stundzia and Charles J. Lumsden. Stochastic simulation of coupled reaction–diffusion processes. *Journal of Computational Physics*, 127:196–207, 1996.

[23] M. Ander, P. Beltrao, B. Di Ventura, J. Ferkinghoff-Borg, M. Foglierini, A. Kaplan, C. Lemerle, I. Tomás-Oliveira, and L. Serrano. SmartCell, a framework to simulate cellular processes that combines stochastic approximation with diffusion and localisation: analysis of simple networks. *Systems Biology*, 1(1):129–138, June 2004.

[24] S. Schnell and T. E. Turner. Reaction kinetics in intracellular environments with macromolecular crowding: simulations and rate laws. *Progress in Biophysics & Molecular Biology*, 85:235–260, 2004.

[25] Otto Berg, Måns Ehrenberg, and Martin Lovmar. *Biophysical Chemistry: Thermodynamics and Kinetics*. Uppsala University, October 2001.

[26] Otto G. Berg. On diffusion-controlled dissociation. *Chemical Physics*, 31:47–57, 1978.

[27] Yoshiki Kuramoto. Effects of diffusion on the fluctuations in open chemical systems. *Progress in Theoretical Physics*, 52:711–713, August 1974.

[28] F. Baras and M. Malek Mansour. Microscopic simulations of chemical instabilities. *Advances in Chemical Physics*, 100:393–474, 1997.

[29] Thomas Fricke and Dietmar Wendt. The Markoff-Automaton: A new algorithm for simulating the time evolution of large stochastic dynamic systems. *International Journal of Modern Physics C*, 0(0):1–29, 1995.

[30] Maria Andér. SmartCell – a general framework for whole-cell modeling and simulation. Master's thesis, Uppsala University School of Engineering, October 2002.

[31] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novère, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[32] Ben Bornstein. *libsbml Developer's Manual*. The SBML Team, 2.3.0 edition, May 2005.

[33] Andrew Finney and Michael Hucka. Systems biology markup language (SBML) level 2: Structures and facilities for model definitions. Technical report, Systems Biology Workbench Development Group, June 2003.

[34] Johan Hattne, David Fange, and Johan Elf. *MesoRD User's Guide*. Uppsala University, Department of Cell and Molecular Biology, 0.2.1 edition, 2006.

[35] Clark Cooper. Using Expat. *O'Reilly XML.com Archives*, pages 1–3, September 1999.

[36] The Apache Software Foundation. *Xerces-C++ Documentation*. The Apache Software Foundation, 2004.

[37] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, third edition, 1997.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley, first edition, 1995.

[39] Karen Lipkow, Steven S. Andrews, and Dennis Bray. Simulated diffusion of phosphorylated CheY through the cytoplasm of *Escherichia coli*. *Journal of Bacteriology*, 187(1):45–53, January 2005.

[40] Carl Nordling and Jonny Österman. *Physics Handbook for Science and Engineering*. Studentlitteratur, fifth edition, 1996.

[41] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[42] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification*. Silicon Graphics, Inc., 1.5 edition, 2003.

[43] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach with OpenGL^{TM}*. Addison–Wesley, first edition, 1997.

[44] Paula Womack and Jon Leech. *OpenGL® Graphics with the X Window System®*. Silicon Graphics, Inc., 1.3 edition, October 1998.

[45] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, first edition, 1996.

[46] Peter Burger and Duncan Gillies. *Interactive Computer Graphics: Functional, Procedural and Device-Level Methods*. International Computer Science Series. Addison–Wesley Publishing Company, first edition, 1989.

[47] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics – Principles and Practice*. The Systems Programming Series. Addison–Wesley Longman, second edition, 1996.

[48] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison–Wesley Publishing Company, first edition, 1996.

[49] Aristides A. G. Requicha and Robert B. Tilove. Mathematical foundations of constructive solid geometry: General topology of closed regular sets. Technical report, College of Engineering & Applied Science, The University of Rochester, March 1978.

[50] Johan Hattne, David Fange, and Johan Elf. Stochastic reaction–diffusion simulation with MesoRD. *Bioinformatics*, 21(12):2923–2924, 2005.

[51] David Fange and Johan Elf. Noise-induced Min phenotypes in *E. coli*. *PLoS Computational Biology*, 2(6):637–648, June 2006.

[52] Caroline Lemerle, Barbara Di Ventura, and Luis Serrano. Space as the final frontier in stochastic simulations of biological systems. *FEBS Letters*, 579:1789–1794, 2005.

[53] Nicolas Le Novère, Benjamin Bornstein, Alexander Broicher, Mélanie Courtot, Marco Donizelli, Harish Dharuri, Lu Li, Herbert Sauro, Maria Schilstra, Bruce Shapiro, Jacky L. Snoep, and Michael Hucka. BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Research*, 34:D689–D691, 2006.

[54] Andrzej M. Kierzek. STOCKS: STOChastic Kinetic Simulations of biochemical systems with Gillespie algorithm. *Bioinformatics*, 18(3):470–481, 2002.

[55] Anders Kaplan. On whole-cell modelling and simulation. Master's thesis, Uppsala University School of Engineering, October 2001.

[56] G. Korniss, M. A. Novotny, and P. A. Rikvold. Parallelization of a dynamic Monte Carlo algorithm: a partially rejection-free conservative approach. Technical report, Florida State University, July 2004.

Appendix A

Algebraic Simplification Transformations

In the diagrams that follow, t_i denotes a subtrees and c_i is a constant node. Constant nodes with known values are indicated by their value alone.

The general tendency of the transformations is to move constant nodes upwards and to the left. If an AST is subjected to these transformations from the leafs and towards the root, in the order given here, the tree *should* eventually be algebraically simplified. The transformations are grouped by the operation defined in root of the subtree.

Caveats apply: this list of transformations needs to be verified. It may be possible to reduce an expression further even after the application of the material in this appendix. Some transformations may be redundant or even erroneous.

A.1 Addition Transformations

Addition transformations are shown in Fig. A.1.

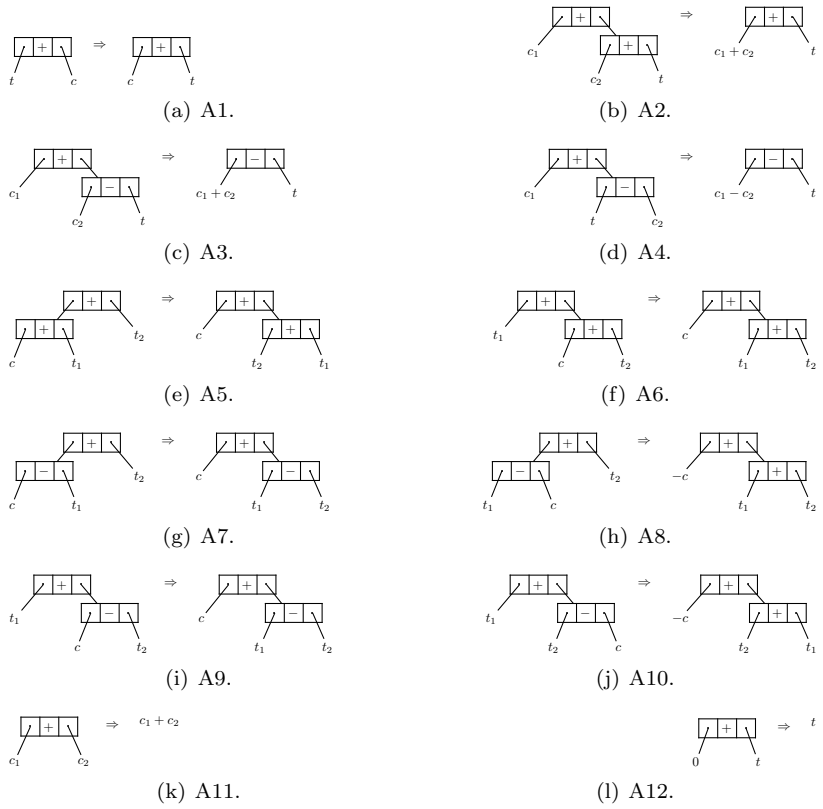


Figure A.1: The addition transformations.

A.2 Division Transformations

Division transformations are shown in Fig. A.2.

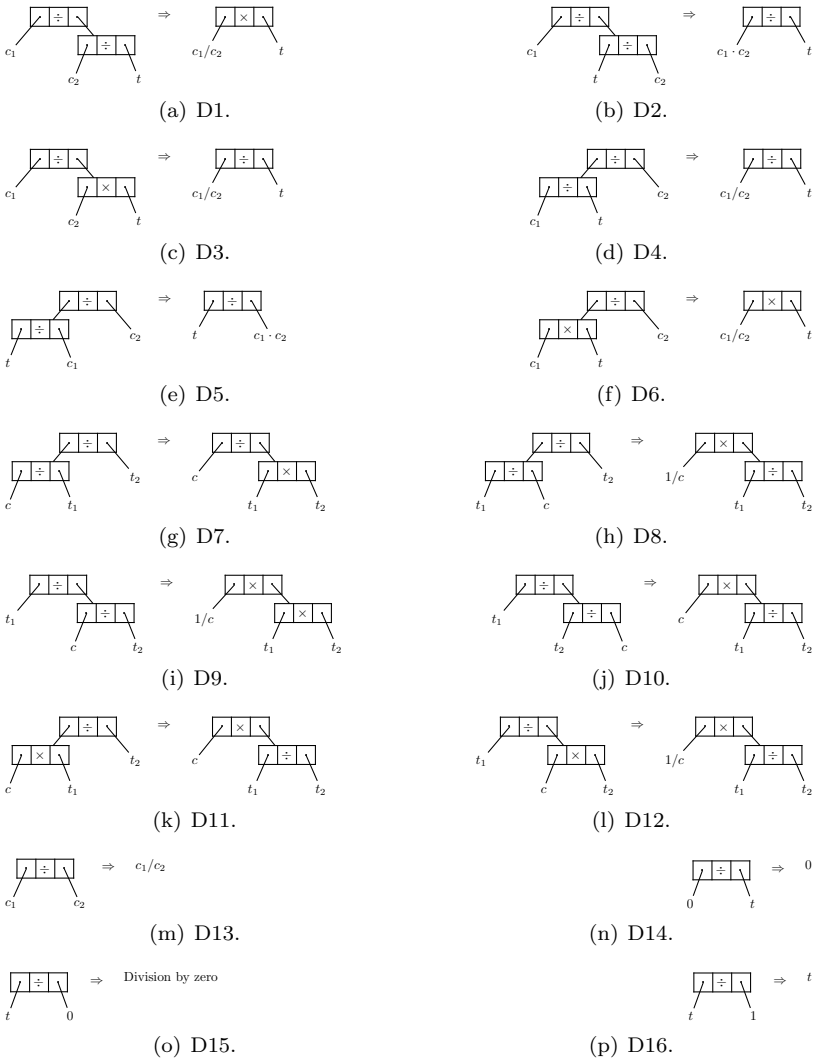


Figure A.2: The division transformations.

A.3 Multiplication Transformations

Multiplication transformations are shown in Fig. A.3.

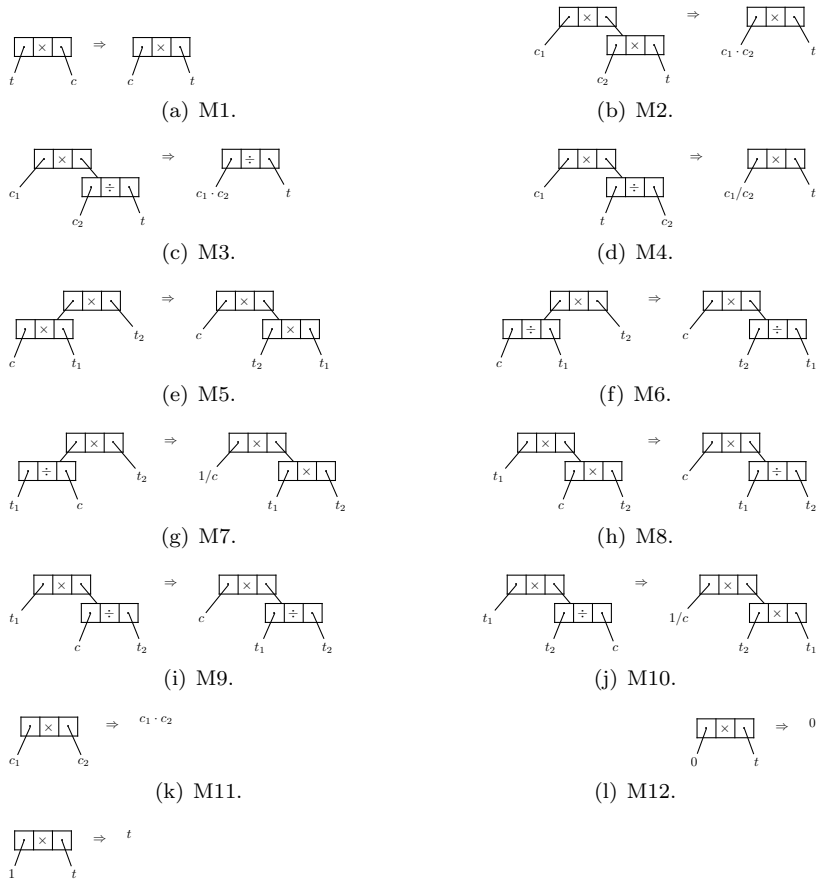


Figure A.3: The multiplication transformations.

A.4 Subtraction Transformations

Subtraction transformations are shown in Fig. A.4.

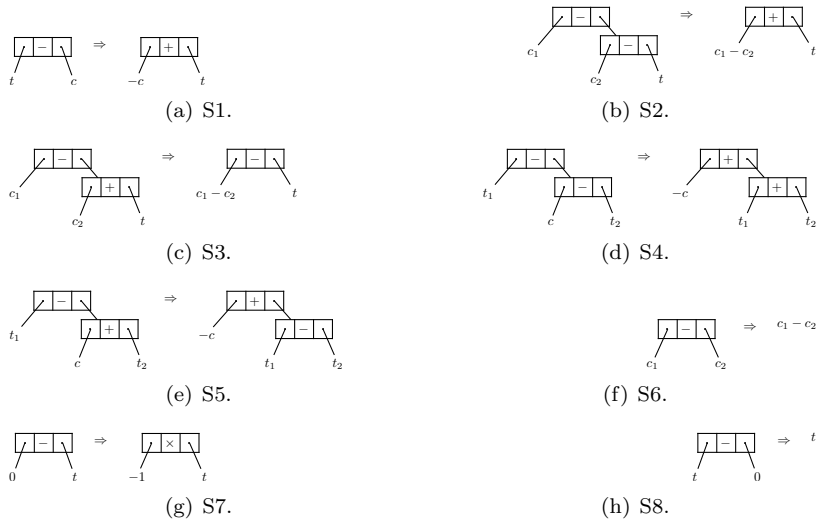


Figure A.4: The subtraction transformations.