GOKMC: Generic Programming Approach to Object Kinetic Monte Carlo

U. Bhardwaj[a], M. Warrier[a]

[a] *Computational Analysis Division, BARC, Visakhapatnam, Andhra Pradesh, India - 530012*

Abstract

We present an object kinetic Monte Carlo (OKMC) library using generic programming with C++ that helps model a problem by static composition of loosely coupled components. The light weight static abstractions incur minimal runtime overhead. New components and strategies such as a process, neighbour search policy, process selection criteria can be added and composed together, given the type requirements are met at compile time. The library can be used to simulate different problems without loss of efficiency. We provide a detailed generic approach to rejection–free KMC (rfKMC) algorithm along with the different policies it depends on. We carry out benchmarks on three different diffusive reactive problems comparing possible design choices and abstractions. The generic approach of the library facilitates changing each component independently which also makes benchmarking and comparing different implementations easy. The benchmarks are shown to highlight performance trade–offs of static abstractions against runtime polymorphism. We validate the library by comparing the results with the existing results.

1. Introduction

Kinetic Monte Carlo (KMC) is a method to simulate time–evolution of a system governed by the Poisson processes. Many natural processes can be simulated with KMC. It is applicable to various fields ranging from social, economical and biological models to radiation damage modeling in materials [1, 2, 3, 4, 5, 6]. The KMC methods are very useful for scaling up the material damage simulations

without losing spatial correlations [7, 8]. KMC carried out only for objects of interest such as defects in an irradiation damage simulation while ignoring the background objects such as lattice atoms is termed as Object Kinetic Monte Carlo (OKMC). The KMC method can be carried out using rejection KMC (rKMC) or rejection–free KMC (rfKMC) algorithms [9]. The basic KMC is an inherently serial algorithm i.e. only one event takes place in a single time–step making it computationally expensive especially for bigger simulation sizes [10]. The algorithms such as [11, 12] alleviate this limitation and thus enable execution of multiple events in parallel. The algorithm shown in [11] limits the number of events to the number of sub–domains which is equivalent to the number of processes in the parallel message passing execution. The computational advantage of the algorithm is shown to saturate as the number of processes increase due to increase in communication between the processes. The algorithm in [12] overcomes it and the number of events in a single time–step can be same as the number of particles. In this work we focus on OKMC with rfKMC algorithm applied to material science problems. Nevertheless, the components and methods discussed can also be reused with other related algorithms and simulations.

The KMC simulation problems share a fair amount of high level logic, however they can largely differ in details. Some of the areas of the variations are as follows:

1. There can be different kind of processes such as jump process for a single particle of specific type or process that adds multiple particles to the whole system.
2. The rate calculation of a process can be independent of state of the system or depend on global state as well as properties of a particle it is applied on [13, 14].
3. The system itself can be a 2D surface or 3D volume with particles that can be defined by a single centroid only or those that also require a shape or multiple points or sizes [15].
4. There can be different kind of spontaneous events or reactions in the system such as recombination of neighbouring particles below a cut–off distance and different kind of boundary conditions for

different directions.

5. The behaviour related to observing the simulation, logging the results etc. may also differ.
6. In addition, a neighbour search criteria [16, 17], process selection strategy, updating and calculating process rate etc. may have multiple possible implementations and one implementation might be more efficient for certain type of simulation than the other.

Many options for different components lead to an exponential number of possible combinations. In the present work we try to model a KMC problem as a composition of small, mostly orthogonal components and implement the components without any extra assumptions using principles of generic programming [18, 19]. The components can be incrementally specialized for a specific problem by non intrusively wrapping and composing the logic around the general cases. The concrete components are statically composed using policy based design [20] which enables selecting, mixing and matching different concrete components (termed as policy implementations). The static composition and use of types to express constraints in the implementation helps in catching more errors at compile time, while also keeping the abstraction cost to minimum [21].

The process selection and process rate calculation can be a limiting factor for the efficiency of rfKMC algorithm if the number of processes are large or rate calculation for a process is non–trivial. There are different strategies to improve efficiency in such cases [13, 14, 22]. The use of observer pattern implemented as system hooks in the library helps in adding these strategies to a simulation if required. Hooks also help in implementing neighbour search, reactions etc. The generic approach and implementation of different policies can also be used for these alternate KMC algorithms in addition to the basic rfKMC algorithm that this work focuses on.

We simulate different problems using the library and validate the results by comparing with existing results. We carry out detailed benchmarks with different possible design choices and abstractions. The

generic approach of the library facilitates switching between the different abstractions for a component. The existing KMC frameworks [15, 23] model KMC problem with object oriented design and runtime polymorphism. In addition to runtime polymorphism, the library allows static light weight abstractions which can result in significant improvement in execution time compared to runtime polymorphism [21] as we show in the benchmarks section.

In Section 2 we discuss the background and preliminaries of generic programming and rfKMC algorithm along with the components it involves. Section 3 gives a formal description of the various components. Section 4 discusses the example problems and the simulation results obtained. Section 5 shows performance benchmarks, comparisons and trade–offs of different abstractions that can be used to model the example problems. Finally Section 6 concludes discussing significance of the presented approach.

2. Background and Preliminaries:

2.1. Generic Programming and Policy Based Design

Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency [21]. Generic programming can be achieved using macros, void pointers and templates. Among these techniques, templates are structured, type–safe at compile–time and they bind statically. Templates allow functions or classes to have types as compile time parameters. The template code generates the needed function or class depending on the types passed to it.

In the policy based design [20] the orthogonal functionalities of a host class are abstracted in different template classes each implementing their own interface (called policy). An exponential number of combinations can be supported by instantiating the host template class with different implementations of the

policies, independently. A policy based class can be extended in unforeseen new ways by implementing new policies.

The library relies on C++ templates for generic programming to implement the rfKMC algorithm and related components. The dependencies of the algorithm are abstracted as policies.

2.2. rfKMC Algorithm

The rfKMC algorithm is also known as Bortz–Kalos–Leibowitz (BKL), n–fold, or residence time algorithm [9, 24]. Following is the pseudo–code for the algorithm.

1. Set the time t = 0.
2. Using the list of all possible transition rates in the system, one for every N number of Poisson processes, calculate the cumulative function $R_i = \sum_{j=1}^{i} rate_j$, for $i \in [1, N]$. The total rate is $Q = R_N$.
3. Get a uniform random number $u \in [0, 1]$.
4. Find the process to carry out by finding the process index i for which $R_{i-1} < u \times Q \leq R_i$.
5. Carry out process i and update the current state.
6. Get a new uniform random number $u' \in [0, 1]$.
7. Increment the time by $\Delta t = Q^{-1} \ln(1/u')$.
8. Check if the simulation has reached culmination, if not then return to step 2.

Following are the details of the terms used in the algorithm. These terms also relate to the orthogonal policies that can be implemented as different components with very low coupling and high cohesion.

- Processes: A process in the KMC simulation is a Poisson processes that alters the state of the system when executed. Poisson processes execute stochastically with a certain rate. KMC requires that the

system evolves via independent Poisson processes. The example of processes in a KMC simulation are: a process making a defect jump with rate depending on the migration energy, a process that adds a number of defects in a lattice or a process that adds a number of people to a queue with a certain rate. Process and system need to be compatible. A process aimed at material simulation can not work with the system that has people and queues in it and vice–versa. A process can be added, removed or its rate can change depending on the state of the system, e.g. in a simulation of radiation damage experiment in which the irradiation is carried out for a certain time–period and then thermal diffusion of defects (that occurs even after irradiation stops) is observed, we will need to remove addition process after certain time–period in the simulation.

- Reactions: A process executes on a system by changing its state. The changes introduced by a process might get the system in a state from which it spontaneously changes to a new state. A system may have a number of behaviours that define the reactions or spontaneous changes that might be required after a change in the system. The examples of reactions include annihilation or recombination of an interstitial and vacancy, boundary conditions etc.

- System: A system in OKMC simulations consists of global system properties such as temperature, dimensions and a collection of different particles. These particles can be of different species e.g. in a FeCu alloy simulation, there can be defects consisting of Iron single interstitial, Copper single interstitial, single vacancy and vacancy cluster etc. A process or reaction is generally applicable for a particular species, e.g. a jump process with certain migration energy may only apply to a particular species. We usually represent the species with a "sum" algebraic data type such as C++ enum. Some particles may also differ in properties e.g. a cluster of interstitials requires a size property to define how many interstitials have combined to form the cluster while a single interstitial does not, a cluster of vacancies and solutes requires size property as well as number of solute. The particles

that differ in the properties are represented by different C++ types, in addition to having different species. Thus, particle types are different C++ types while species are values of the same type. We define systems that can be composed of multiple types of particles either using "product" algebraic data type such as C++ tuple or runtime polymorphism with inheritance.

- Predicate: Predicate is any function that returns a boolean. The rfKMC algorithm requires a predicate for the termination condition, which is checked at every step. Usually, the predicate depends on the current system state and simulation time. It may also depend on the number of steps which is same as the number of times predicate has been called by the algorithm.

3. Formal Definitions and Implementation

3.1. rfKMC Algorithm

We implement rfKMC algorithm as a template function. Similar to the algorithm in pseudo–code, the implementation does not assume anything about the nature of the system, processes etc. beyond what is specified in the algorithm. The algorithm specifies orthogonal components as template parameters following a policy based design. The components passed to the algorithm are supposed to provide an interface to be used. Following are the policies along with the interface requirements.

- ProcessCollection:
  - cumulativeRates(System): returns iterable and indexable list / container [18] of type double e.g. std::vector, std::array etc
  - executeSelected(int processIndexToExecute, System): an action [18] that executes the selected process at the input index.

- Random: procedure (non–regular) with no input parameters having result definition space = $[0.0, 1.0]$
- Predicate: predicate that takes system and time as input parameters.
- System:
  - update(): Signals completion of the process event for the current step.

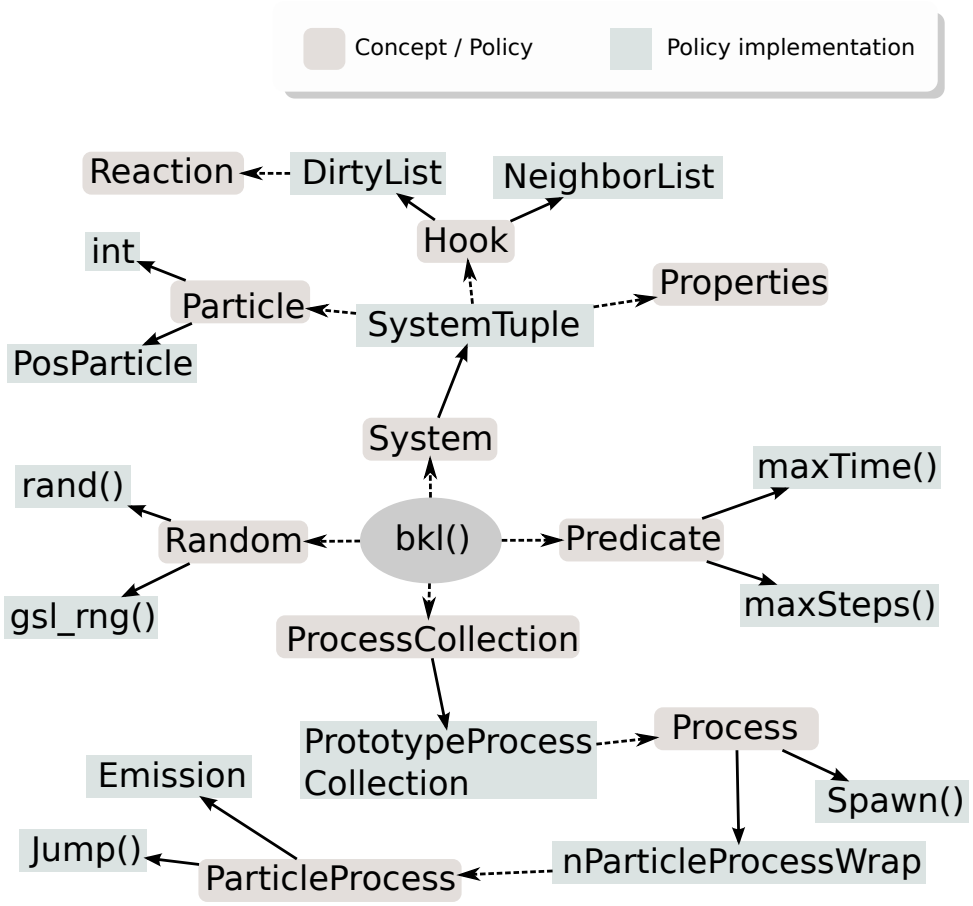

Figure 1: The figure shows policies and some of their implementations. The implementations with parenthesis at the end are functions.

More details of the above policies are discussed later in the section. The generic rfKMC algorithm is given below. The corresponding step number from the pseudo–code given in Subsection 2.2 is indicated in the comments at the end of the lines.

```
enum class bklStatus : bool {NoProcesses, Finished};
template <class ProcessCollection, class Random, class Predicate, class System>
auto bkl(ProcessCollection &&procs, Random &&rnd = Random(),
         Predicate &&predicate = Predicate(), System &&sys = System()) {
  auto systemTime = 0.0;  // 1
  while (predicate(sys, systemTime)) {
    auto rates = procs.cumulativeRates(sys); // 2
    if (rates.empty() || rates[rates.size() - 1] < epsilon) {
      return make_pair(systemTime, bklStatus::NoProcesses);
    }
    auto urn = rnd() * rates[rates.size() - 1]; // 3
    auto picked = std::lower_bound(begin(rates), end(rates), urn); // 4
    auto processIndex = std::distance(begin(rates), picked);
    procs.executeSelected(processIndex, sys); // 5
    sys.update();
    systemTime += std::log(1.0 / rnd()) / (rates[rates.size() - 1]); // 6, 7
  }
  return std::make_pair(systemTime, bklStatus::Finished);
}
```

To simulate a problem, we call the above function with the corresponding policies configured according to the problem. In addition to the interface requirements of the algorithm, the policies that are inter–dependent also need to fulfill the requirements imposed by each other, e.g. the processes for material damage simulation might require that the system provides certain kind of interface to move and alter particles in addition to the update() interface required by the rfKMC algorithm. The interface requirements on types are checked by the compiler during compile time. However, for the correct implementation of the algorithm more requirements need to be satisfied such as the condition that the Random procedure must have the result definition space of [0.0, 1.0]. These can be expressed as preconditions using types and runtime checks, e.g. the definition space for Random procedure can be made as a necessary requirement by having a 'validateUrn' type that gets instantiated only if the real number is between the range. By making sure that rest of the functions take as input the validated type rather than usual primitive type, compiler can assure that the validation checks are always present. The runtime validations incur some overhead. We may not explicitly mention the preconditions and validations in the discussion, however they are necessary tools to reason the correctness of the code.

The use of policy based design makes the design logic simpler and provides a way to express the orthogonal aspects of the code, making different components easy to understand and compose together. In addition, the policies facilitate unit testing e.g. to test the process selection step of the algorithm on the edge cases i.e. when the random number for process selection returns exactly 1.0 or 0.0, we can pass a dummy function in place of random number generator that always returns 1.0 or 0.0 and check if the result is as expected. Similar, test policies can be used for unit–testing different aspects independently.

3.2. ProcessCollection

A class that provides the following interface can be used as a process collection policy.

- cumulativeRates(System): returns iterable and indexable container / list of cumulative rates of all the processes.
- rate(System): returns the total rate of all the processes.
- executeSelected(int processIndex, System): executes the process that is indexed by input parameter processIndex.
- execute(System): selects a process and then executes it.

A ProcessCollection policy needs to execute different behaviour based on the process selected which can be implemented using a conditional switch like control flow. However, this will require a different ProcessCollection for each problem. A ProcessCollection that can be composed of different processes implementing different behaviours allows for more flexibility and reuse. The interface requirements for a process that can be added to the collection is as follows:

- execute(System): the action that changes the state of the system.
- rate(System): returns the rate of the process.

Since, both the interface functions required for a process are also part of the ProcessCollection policy, a ProcessCollection can be contained in another ProcessCollection. The self composition is useful to compose different ProcessCollections hierarchically. A ProcessCollection can thus have a subset of processes and may have a specialized way of selecting the process or calculating cumulative rates. The binning or grouping criteria for calculating cumulative rates such as described here [25] can be implemented using the self composition of ProcessCollections.

The addition of different processes in the collection can be implemented using static or runtime polymorphism. We implement following two process collections:

1. TupleProcessCollection: The TupleProcessCollection contains a tuple (a product type) of different types of processes. Any class that provides the required interface for a process can be added. Following is the template class body along with the declaration of interface functions.

```
template <class System, class... Procs>  struct TupleProcessCollection {
  const array<double, sizeof...(Procs)>  &cumulativeRates(System &sys);
  void execute(System &sys);
  void executeSelected(size_t i, System &sys);
  double rate(System &sys);
};
```

2. PrototypeProcessCollection: The PrototypeProcessCollection can contain different type of processes by internally using runtime polymorphism [26]. Same as TupleProcessCollection, any class that provides the required interface for a process can be added.

The processes such as introduction of particles to the system or change in some global property of the system apply to the whole system while others such as jump, emission apply to each particle of a species. The rate of a particle process might be same for all the particles or may depend on the properties of the individual particle. This distinction affects total rate calculation for the selection of the particle for execution of the process. The execute function of a particle process takes a particle as input in addition to the system. We define following two wrappers for the particle processes.

1. nParticleProcessWrap: The nParticleProcessWrap wraps a single particle process which has same rate for every particle of a species. The wrapper returns the total rate that equals the rate of the process for single particle multiplied by the number of particles of that species in the system. When executing, a particle of the species is chosen randomly and passed to the particle process for

execution. The rate calculation and process selection implemented in this way is efficient but it only applies if the rate of the process is same for all the particles of a species.

2. PropParticleProcessWrap: This wrapper is similar to nParticleProcessWrap, however it is applicable to particle processes where the rate of the particle process depends on the individual particle properties e.g. the rate of emission process may depend on the size of the cluster, resulting in different rates for two particles having same cluster species but different sizes. The rate calculation and process selection is not as efficient as in the nParticleProcessWrap in the general case, however it can be optimized for some specialized cases by using system hooks that are discussed in the next subsection.

3.3. System

A system consists of global properties such as temperature, dimensions etc. and a number of particles of different types and species. With the time evolution of the rfKMC algorithm different processes acting upon the system change the global properties and particles. The System needs to provide an interface to access and change global properties, and for addition, removal and traversal of the particles.

To further extend the system, hooks can be attached to it as an observer. Whenever the system changes, it calls the attached hook e.g. if a particle is added to the system, hook.added(particleId) is called, if a particle is edited hook.edited(oldParticle, particleId) is called etc. Hooks can be used to update cell linked list for neighbour search, logging results, process rates in a particle process collection such as PropParticleProcessWrap, as proposed in [13, 14] etc.

A System can be implemented with just a single particle type for problems in which all the particles have same properties. Static and runtime polymorphism can be used to model problems that involve particles with different set of properties.

It is always possible to represent all the particles with a single type that contains all the properties required by any of the species. Some of the properties might not be well defined for all of the species e.g. size and number of solute have well defined meaning for a cluster while for a single interstitial both of the properties have no meaning. In this case, the real–world states do not have a one to one mapping to the states represented in the code, leaving some of the values as redundant and some as not well defined. This may result in errors [27], less expressive code and difficulty in reasoning about the correctness of the code.

Following are the different implementations of System policy that we implement and benchmark.

1. SystemTuple: A particle in the SystemTuple is characterized by its type, species and index number. The template class can be initialized with different particle types implemented with the help of a tuple. Following is the definition of the SystemTuple with template parameters.

   ```
   template <class SystemProps, class Hook, class... Particles>  class SystemTuple;
   ```

2. SystemSingle: It is similar to SystemTuple except that it can only have one type of particle. We show the performance comparison of SystemSingle and SystemTuple for the problems that only need single type of particle, in next section (Section 5).

   Following is the definition of the SystemSingle with template parameters.

   ```
   template <class SystemProps, class Hook, class Particle>  class SystemSingle;
   ```

3. SystemPtr: The particles are allocated on heap and the pointer to the memory of the particles is stored in the system. More concretely, we use vector of pointer of particle type rather than using vector of particle type directly. It can have multiple types of particles by using runtime polymorphism. When runtime polymorphism is used to add multiple types of particles, it can have

additional overhead due to presence of virtual table and dynamic dispatch. We show the detailed comparisons in the benchmarks section.

System takes a single hook only, however two or more observers might be needed at the same time. For this we define a simple hook 'Both' which takes two other hooks as template parameters and calls them both for any change in the system. Using Both we can attach multiple hooks to the system e.g. three hooks can be attached by adding Both<Both<hook1, hook2>, hook3>.

We also define a number of basic particles. The PosParticle only contains an array of coordinates for the position of the particle. The SizeWrap and PbcWrap contain an integer size and an array of periodic boundary counts, respectively. The array of counts is useful to unroll the coordinates of a particle if periodic boundary conditions apply. The SizeWrap and PbcWrap use CRTP (Curiously Recurring Template Program) [20] so that they can be wrapped around any particle type e.g. SizeWrap<PosParticle>particles have a centroid and a size value, SizeWrap<PbcWrap<PosParticle>>particles have a centroid, size and PBC counts array.

If a simulation requires particles with only single integer then we can create a system with a C++ *int* particle type such as $SystemTuple < SomeProps, SomeHook, int >$. Unlike runtime polymorphism, the particle type need not inherit from an interface. If the processes and reactions apply operations to the particles that are well defined for an *int* type, the simulation will compile fine else there will be compile time type errors.

3.4. Reactions

The reactions execute on the particles that have been added or edited by the events in the current step of the algorithm. We term these particles as dirty particles. A process or reaction event can result in multiple dirty particles e.g. a process can introduce multiple particles, an emission process changes size

of a cluster as well as adds another particle, an annihilation reaction between a vacancy–solute cluster and a bigger cluster of self interstitials may end up with solute particles and an interstitial cluster of smaller size.

A reaction has a predicate that checks if it needs to act on a dirty particle or not, e.g. a periodic boundary condition (PBC) reaction only executes if the particle has moved outside of the boundaries of the system. A reaction might only be applicable to a certain species. A reaction can either be called as soon as a particle is dirtied (eagerly) or it can wait until the process or reaction makes all the changes, returns and system is updated. We implement both the schemes as hooks to the system as given below:

- DirtyList: It takes a reaction type as template parameter. It keeps track of all the dirty particles as it observes the system. When rfKMC calls for system update, the system hook is notified to update as well. On update the DirtyList checks for reaction execution on each of the dirty particles. If the reaction does not occur for a dirty particle, it is removed from the dirty list. If the reaction occurs it changes the system and in turn can add to the dirty list of the particles. The update procedure ends only when the dirty list is empty. A process or reaction might call system update if it wishes to execute reactions and then continue with more changes in the system.

- Eager: On any change in the system Eager checks if the input reaction executes for the particle. If the reaction occurs, then it may change the system, which in turn recursively calls Eager.

Both Eager and DirtyList require a single reaction function procedure that returns true if it executes for an input particle and false if it does not. Although, both the hooks take only a single reaction, a problem usually requires multiple reactions. Multiple reactions can be combined with small reaction combiner classes and conditionals which themselves implement the same interface as a reaction for use with Eager or DirtyList.

Some of the reaction combiners are described below:

- Fallback: The type takes two reaction types as template parameters. When executed, it calls the first reaction, if the first reaction returns true, the combiner also returns true. If the first reaction returns false only then the second reaction is executed and its return value is returned. The implementation is given below:

```
template <class R1, class R2> struct Fallback {
  Fallback(R1 r1, R2 r2) : _r1{r1}, _r2{r2} {}
  template <class Sys, class Particle>
  auto operator()(Sys &a, Particle &b) {
    if (_r1(a, b)) return true;
    return _r2(a, b);
  }
private:
  R1 _r1;
  R2 _r2;
};
```

  Multiple Fallbacks can be nested to combine more than two reactions e.g. $Fallback < R1, Fallback < R2, R3 >>$ can be used to combine three reactions. The same can be written as $Fallback < R1, R2, R3 >$ which is just a syntactic sugar that works for any number of reactions.

- TypeSelect: It takes two reaction types and a particle type as template parameter. If the template particle type is same as type of the input particle then the first reaction executes else the second reaction executes. The selection is made at the compile time.

- PredicateWrap: It takes two reactions and a predicate as input. If the predicate return true for a particle then the first reaction executes else the second reaction executes. The second reaction if not given, defaults to a NoReact reaction which always returns false.

- SpeciesSelect: It takes two reactions and a collection of species as input. It is a specialized type of PredicateWrap in which the predicate returns true only if the collection of species contains the species of the input particle.

In place of the combiners mentioned, we can have a ReactionCollection policy, which can contain all the reactions for the different species by using runtime polymorphism. We compare the performance of the implementations in the Section 5.

We have defined reactions for a single dirty particle, however a lot of the reactions occur between two particles that are in close vicinity of each other e.g. annihilation or absorption reactions. We implement binary reaction wrappers that themselves are normal reactions but take as input a binary reaction procedure. A binary reaction is same as a reaction but takes two particles as the input parameter instead of one. If no particle is within the cut–off radius of the input dirty particle then it returns false else it executes the binary reaction with the neighbour and returns true. Some of the types of binary reaction wrappers are:

- ReflexiveType: The binary reaction occurs only if both the particles are of same type. The decision to execute the binary reaction or not for any given particles is made at the compile time.

- BinaryPredicateWrap: The Wrapper takes a predicate that takes in two particles. If the predicate returns true then only the binary reaction occurs. All the following wrappers are specialized type of this.

- Reflexive: The binary reaction occurs only if both the particles are of the same species.

- Symmetric: The binary reaction occurs between the members of two given set of species only.
- Any: The binary reaction occurs between any of the given species of particles.

The reaction combiners along with binary wrappers provide a myriad of possibilities to create reactions e.g. a reaction: Fallback<PBC, SpeciesSelect<A, Reflexive<Recombine>, NoReact>>expresses that first PBC is checked for execution which only executes if a particle is outside the boundary. If the PBC does not execute and the species of the particle is A, neighbours of the particle having same species A are searched, if found the recombine reaction executes. If no such neighbouring particle is found no reaction executes.

The binary wrappers require the closest neighbouring particle that is within the cutoff radius of the input dirty particle. It abstracts away the neighbour search criteria as a template parameter. We implement two types of neighbouring criteria, brute-force search and cell linked list search [16]. The cell linked list also requires to be added as a hook to the system since it needs to update internal data structures according to change in coordinates of the particles. Since, the neighbour search is abstracted as a policy, a combination of multiple criteria or more sophisticated criteria [17] can be used easily.

4. Example Problems

We present the example problems used for validation and benchmarks along with the simulation parameters and short description. The results obtained are compared with the available results for validation of our implementation. The implementation details of policies for neighbour search, randomization etc. does not affect the results obtained, although they do change the performance characteristics. The C++ code for the first simulation is given in Appendix.

4.1. Hydrogen Absorption–Desorption

The simulation models adsorption, desorption and diffusion of hydrogen on a 2D smooth surface [24, 28]. The surface adsorbs the incoming hydrogen atoms which then diffuse on the surface. When two atoms come closer than a cutoff radius, they combine to form a hydrogen molecule. The processes of adsorption of hydrogen atoms and desorption of hydrogen atoms and molecules attain a dynamic equilibrium after some time. The simulation parameters are as follows:

- System:
  - SystemProps:
    - dimensions: 2 (2 dimensional surface)
    - species: H, H2 (hydrogen atom and hydrogen molecule)
    - temperature: 600K
    - box size: 1000 x 1000 Angstroms
  - Particles:
    - PbcParticle<PosParticle>
- Reactions:
  - Fallback composed of the following reactions:
    - PBC: periodic boundary conditions.
    - $SpeciesSelect < Reflexive < Addition >> (H, H2)$: Addition reaction for producing H2 from two hydrogen atoms.
  - Recombine length 2.0 Angstroms
- Processes:

- SpawnSingle(H): rate: $10^{24}/m^2/s$. Introduce single hydrogen atoms at given rate.
- nParticleProcessWrap<Jump2DRandom>(H): 2D Jumps for hydrogen atoms with equal rate for each particle.
  - migration energy: 0.9 eV
  - attempt frequency: 10^{13}
  - jump length: 34.4 Angstroms
- nParticleProcessWrap<Remove>(H): desorption of hydrogen atoms with migration energy of 1.9 eV.
- nParticleProcessWrap<Remove>(H2): desorption of hydrogen molecules with migration energy 0.6 eV.

Fig. 2 shows the time evolution of number of hydrogen atoms on the surface as the simulation progresses. The plot is in agreement with the results shown in [24, 28]. The number of hydrogen atoms on the surface saturate at around 1e-5 seconds in the simulation.

4.2. Displacement Cascade Aging

The problem simulates the thermal diffusion and reactions of the defects caused by a primary knock on atom in a Fe–0.2%Cu lattice [15]. The simulation parameters are taken from Set–B parameters of the [15] in which all the defects including clusters of self–interstitials and clusters of vacancies are mobile except Cu defects. The simulation parameters are as follows:

- System:
  - SystemProps:
    - dimensions: 3

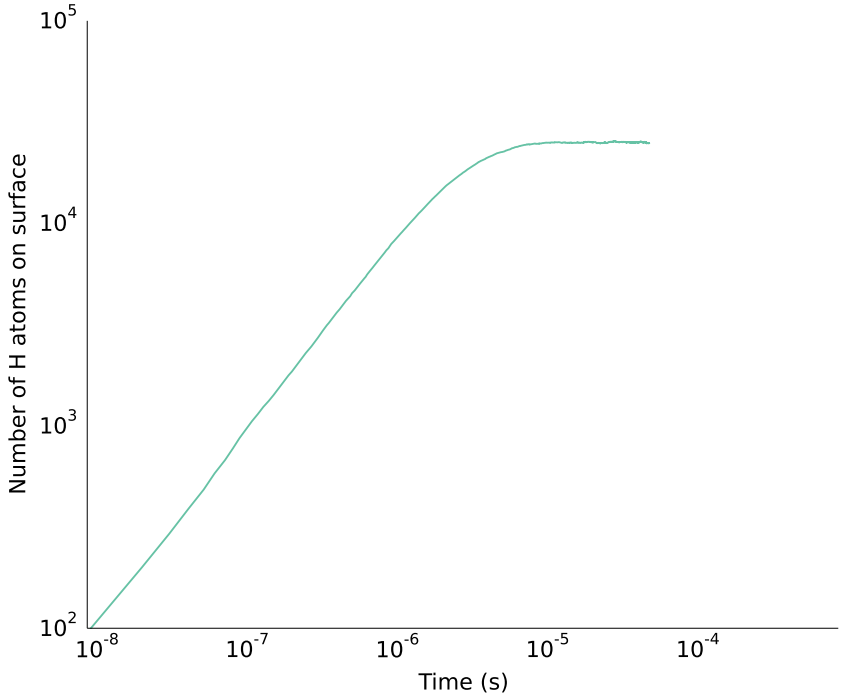

Figure 2: Approach to steady state hydrogen densities on the surface at 600K.

    * species: I, IC, V, VCS, S (SIA, SIA cluster, vacancy, vacancy-solute(Cu)-cluster, Solute)
    * temperature: 600K
  - Particles:
    * $PosParticle, Size < PosParticle >, nSolute < Size < PosParticle >>$
    * defects introduced according to the results obtained with MD simulation of 20 keV pka.

- Reactions:
  - Fallback composed of following reactions:
    * *AbsorbingBC*: Absorbing boundary conditions.

* $Any < Addition > IC, I$: Addition of SIA and SIA clusters.
* $Any < Solute < Addition >> V, S, VCS$: Addition of vacancy, solute and vacancy-solute clusters.
* $Symmetric < Annihilate > I, IC, V, VCS$): Annihilation between interstitials, vacancies and their clusters.

Cut-off radius for binary reactions are as defined by the equations given in [15].

- Processes:
  - nParticleProcessWrap<Jump3DBcc>(I): 3D Jumps for SIA with equal rate for each particle.
  - nParticleProcessWrap<Jump3DBcc>(V): 3D Jumps for single vacancy with equal rate for each particle.
  - PropParticleProcessWrap<Jump1DBcc>(IC): 1D Jumps for SIA clusters with rate depending on cluster size.
  - PropParticleProcessWrap<Jump3DBcc>(VCS): 3D Jumps for vacancy–solute clusters with rate depending on cluster size and number of solute particles.
  - PropParticleProcessWrap<Emission>(IC): Emission of single SIA from SIA cluster with rate depending on cluster size.
  - PropParticleProcessWrap<Emission>(VCS): Emission of single vacancy from vacancy–solute cluster with rate depending on cluster size.
  - PropParticleProcessWrap<EmissionSolute>(VCS): Emission of a vacancy-solute pair from vacancy–solute cluster with rate depending on cluster size.

Fig. 3 shows the time evolution of number of SIA, vacancies and their clusters. Initially, SIA diffusion is the dominant process which happens in fast time–scale compared to other processes. The initial increase

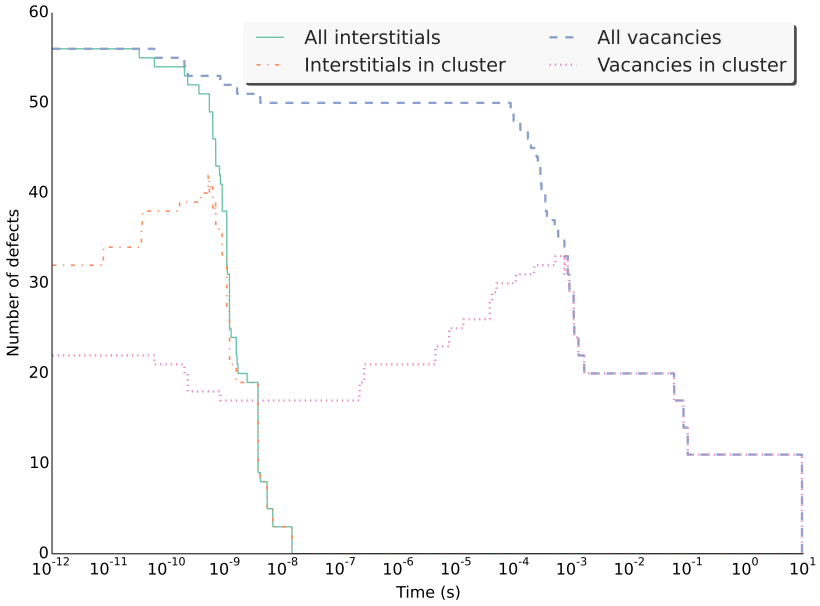

Figure 3: Evolution in time of number of SIA, vacancies and their clusters, during the aging of a 20 keV displacement cascade in Fe-0.2%Cu at 600K

in number of interstitial clusters implies that some of the SIA defects combine to form SIA clusters while decrease in number of vacancies is a result of annihilation of diffusing interstitial by vacancy and vacancy clusters. The vacancies diffusion dominates once all the SIA defects have been absorbed by the boundaries or annihilated. The single vacancies then either form clusters or get absorbed at the boundaries. The diffusion of vacancy cluster which is the slowest of all diffusion then leads to the absorption of all the defects in the simulation. The time–scales of the events and trends obtained in the plot are same as shown in [15].

4.3. Helium Diffusion

The problem simulates helium diffusion in a lattice [29]. The single helium atom defects diffuse and combine to form non–diffusing helium clusters. The helium clusters sometimes dissociate and emit single

helium atoms. The input parameters for the problem are as follows:

- System:
  - SystemProps:
    - dimensions: 3
    - species: He, HeC (Single Helium and Helium cluster)
    - temperature: 800K
    - boxDimensions: 10^5 Angstroms for the results, 10^4 Angstroms for the benchmarks,
  - Particles:
    - Pbc<PosParticle>, Size<Pbc<PosParticle>>: for He and HeC, respectively.
- Reactions:
  - Fallback composed of the following reactions:
    - PBC: periodic boundary conditions.
    - $Any < Addition > (He, HeC)$: Clustering of single helium defects and its clusters.

    Cut-off radius for clustering binary reaction is a function of size of the He defect as given in [29].
- Processes:
  - SpawnSingle(He): Addition of single He atom with a rate of 22 atoms per second.
  - ParticleWrap<Jump3DRandom>(He): Jump in random 3D directions for He diffusion with same rate for all the He atoms.
    - migration energy: 1.257 eV

* attempt frequency: 2.0×10^{12}
* jump length: 9.0 Angstroms

– PropParticleProcessWrap<Emission>(HeC): Emission of single He atom from He cluster.

* rate: function of the size of the cluster

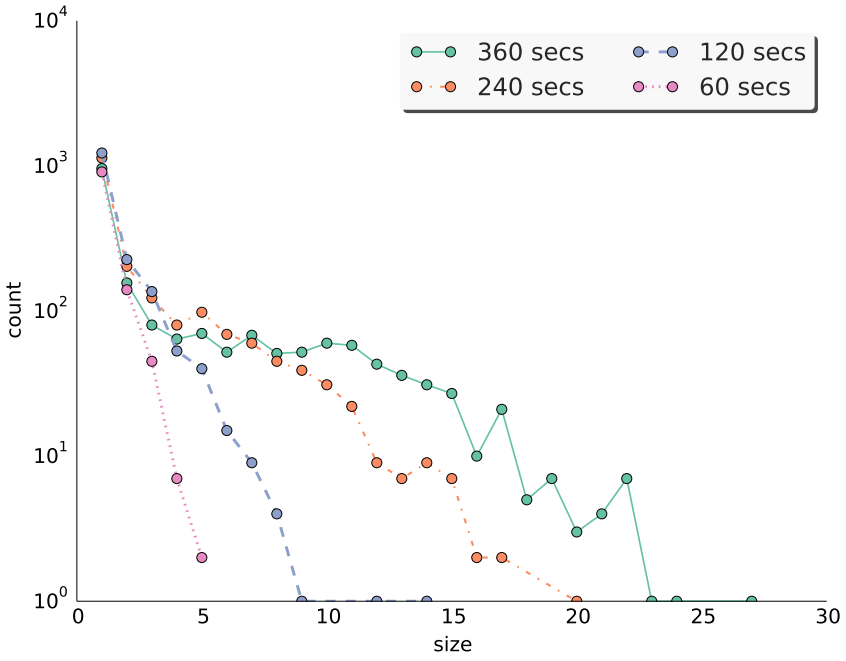

Figure 4: The cluster size distribution of Helium at four different times in the simulation viz. 60s, 120s, 240s and 360s at 800K.

The Fig. 4 shows the cluster size distribution of Helium at 60s, 120s, 240s and 360s in the simulation at 800K. The helium density increases with time by introduction of more helium atoms. The number of single helium atoms increase almost linearly till approximately 120secs after which they start to decrease due to increase in clustering as a result of increased density. With increase in time bigger clusters start to appear more and number of small sized clusters keeps on decreasing.

5. Benchmarks

We show performance comparison of various policies for the example problems. All the simulation codes are compiled with g++–6.1.1 on a Linux operating system.

Following are the policy implementations that we compare:

- System: SystemTuple, SystemSingle, SystemPtr with and without runtime polymorphism.
- Hook: Dirty and Eager.
- Reactions: Combiner (static dispatch) and ReactionCollection with runtime polymorphism.
- ProcessCollection: TupleProcessCollection (static dispatch) and PrototypeProcessCollection (runtime polymorphism).

The hydrogen absorption–desorption is a particularly good problem for benchmarking to get decent statistics on execution time since the number of particles remain high enough and approximately the same once equilibrium is reached. The Table 1 lists execution times with brute–force neighbour search policy while Table 2 lists execution times with cell linked list neighbour search policy. Each of the benchmark is executed ten number of times to get stable statistics.

Table 1: The mean execution times and standard deviation for the hydrogen adsorption–desorption simulation with brute force neighbour search. The simulation is run for 1000000 steps at 500K. The number of particles at equilibrium is approximately 44400.

System	Hook	Reactions	Processes	Mean Exec. Time(sec)	Std. Dev.
tuple	dirty	combiners	tuple	240.629	0.6507
ptr(polymorphic)	dirty	collection	prototype	395.856	0.405
single	dirty	combiners	tuple	237.91	0.2154
ptr(polymorphic)	dirty	combiners	tuple	395.032	0.644
ptr	dirty	combiners	tuple	358.192	0.184473
tuple	eager	combiners	tuple	243.105	0.346
tuple	dirty	collection	prototype	241.67	0.63

Table 2: The mean execution times and standard deviation for the hydrogen adsorption–desorption simulation with cell linked list neighbour search. The simulation is run for 4000000 steps at 500K. The number of particles at equilibrium is approximately 44400.

System	Hook	Reactions	Processes	Mean Exec. Time(sec)	Std. Dev.
tuple	dirty	combiners	tuple	56.1271	0.29
ptr(polymorphic)	dirty	collection	prototype	62.9078	1.257
single	dirty	combiners	tuple	55.4432	0.4186
ptr(polymorphic)	dirty	combiners	tuple	61.2626	1.0925
ptr	dirty	combiners	tuple	60.9202	0.951
tuple	eager	combiners	tuple	56.4413	0.227
tuple	dirty	collection	prototype	56.5758	0.354

We observe that the execution time with heap allocated particles (SystemPtr) is significantly more. The heap allocation and memory indirection leads to poor cache performance. When the particles are stored in contagious memory rather than indirect memory access, the spatial locality reduces cache misses while traversing the particles. The heap allocation is a prerequisite for runtime polymorphism. The runtime polymorphism further slows down the execution due to virtual table and dynamic dispatch. With the static polymorphism the execution time remains very similar to using a system that allows only a single type of particle.

The use of dirty or eager hooks and use of runtime polymorphism or static polymorphism for processes and reactions has no significant affect on the execution times.

The Table 3 shows the execution times for different policies for the helium diffusion problem. The problem shows similar trends as observed for the hydrogen absorption–desorption problem.

Table 3: The mean execution times and standard deviation for the helium diffusion simulation with brute force neighbour search. The simulation is run for 120 seconds of simulation time at 800K. The number of particles at the end of the simulation is approximately 2640.

System	Hook	Reactions	Processes	Mean Exec. Time(sec)	Std. Dev.
tuple	dirty	combiners	tuple	33636.3	1178.74
ptr(polymorphic)	dirty	combiners	prototype	43470.2	1746.12
single	dirty	combiners	tuple	35293.3	1434.97
tuple	dirty	NotCached	NotCached	59742.5	1366.45

In the helium diffusion problem, the recombination and emission rate depend on the size of the helium defect cluster. The recombination radius and emission rate for the different size of the clusters that gets formed in the system can be cached, thus computing the function only the first time for each size. We add a comparison to evaluate the effect of caching the results against computing it every time. Not using caching results in significant overhead. Such optimizations are common in many of the KMC problems.

The Table 4 shows the change in execution times with the change in cell–lengths of neighbour search policy for the hydrogen absorption-desorption problem. The execution-time decreases significantly with decrease in cell–length. However, cell–length can not be decreased below the capture radius in the simple cell linked list implementation. The trend in general may not be as clear as with the given problem having constant particle density at equilibrium.

Table 4: The mean execution time for hydrogen adsorption–desorption problem for different cell lengths of cell linked list neighbour search.

Cell–lengths(Angstroms)	12.5	25	50	100
mean execution–time(sec)	20.7247	56.1271	249.881	821.094

6. Conclusions

We have presented a generic approach to the rfKMC algorithm that allows expressing wide range of problems efficiently with different choices of abstractions. We discuss the generic approach and implementation of static as well as runtime abstractions for the different reusable components in OKMC material simulations. We express three different problems by describing the properties and choice of abstraction for the policies. The results obtained with different configurations for the problems match with the existing published results. The detailed benchmarks show that runtime polymorphism can be computationally expensive for OKMC and related simulations while static polymorphism can be implemented with very minimal runtime overhead. The library following the principles of generic programming expresses the

OKMC algorithm in a very general setting without loss of efficiency. The approach can also be used to generalize other related KMC algorithms. The processes, reactions and other components can be used with other KMC based algorithms that include lattice KMC [30], new parallel OKMC algorithms [12] etc. The current library implementation focuses on flexibility for expressing any KMC problem with very specific optimizations if required which as a side–effect requires client C++ code for every problem. More convenient domain specific applications and user interfaces can be built over the library to allow users to run simulations by composing the already implemented domain specific policies as is.

Appendix A. Application code for Hydrogen Absorption–Desorption problem

```
// problem constants
constexpr auto temperature = 600.0;
constexpr auto boxDim = 1000.0;
constexpr auto recombineLength = 2.0;
constexpr auto w = 1e13; // attempt frequency
constexpr auto spawnRateH = 1e24 * 1e-20 * boxDim * boxDim;
constexpr auto emJump = 0.9;
constexpr auto lenJump = 34.4;
constexpr auto emRemoveH = 1.9;
constexpr auto emRemoveH2 = 0.06;
constexpr auto maxSteps = 1e6;
// System global properties
enum class Species : bool { H, H2 };
using Props = SystemProps<2, Species>; // 2D lattice with Species as H, H2
```

```
auto p = Props{};
p.temperature(temperature);  // setting temperature in system properties
p.boxDimensions(Props::Coords{{boxDim, boxDim}}); // setiing box dimensions
 // Particle with pbc counts & position properties
using Particle = PbcParticle<PosParticle>;
// reactions and hook - addition reaction for H + H -> H2
using ReflexiveAdd = Reflexive<Addition, NeighbourBrute>;
auto HreflexiveAddH = speciesSelect(Species::H,
                                    ReflexiveAdd{recombineLength});
auto reaction = fallback(PBC{}, reflexiveAddH);
auto hook = dirtyList<Props, Particle>(reaction);
// System
auto sys = systemSingle<Particle>(props, hook);
// Processes
auto procs = PrototypeProcessCollection(sys);
// poisson process for introduction of H
procs.add(spawn(sys, spawnRateH, Species::H));
// poisson process for random diffusion of H on the lattice
auto jump = jump2DRnd(sys, emJump, w, lenJump, Species::H);
// jump process is added for all the particles
procs.add(nParticleProcessWrap(jump));
// remove processes for desorption, added for all the particles
procs.add(nParticleProcessWrap(remove(sys, emRemoveH, w, Species::H)));
```

```
procs.add(nParticleProcessWrap(remove(sys, emRemoveH2, w, Species::H2)));
// Predicate, returns false after maxSteps
auto pred = maxStepsPredicate(maxSteps);
// Random number generator
auto rnd = randReal(0.0, 1.0);
// Simulation, returs total time and status: success or no process to exec.
auto timeStatus = bkl(procs, rnd, pred, sys);
```

References

[1] L. Pareschi, G. Toscani, Interacting Multiagent Systems: Kinetic equations and Monte Carlo methods, Oxford University Press, 2013.
URL http://EconPapers.repec.org/RePEc:oxp:obooks:9780199655465

[2] S. Raychaudhuri, Kinetic monte carlo simulation in biophysics and systems biology, in: V. W. K. Chan (Ed.), Theory and Applications of Monte Carlo Simulations, InTech, Rijeka, 2013, Ch. 10. doi:10.5772/53709.
URL http://dx.doi.org/10.5772/53709

[3] M. J. Caturla, N. Soneda, E. Alonso, B. D. Wirth, T. D. de la Rubia, J. M. Perlado, Comparative study of radiation damage accumulation in cu and fe, Jnl. Nucl. Mater. 276 (2000) 13–21.

[4] C. Björkas, K. Nordlund, M. J. Caturla, Influence of the picosecond defect distribution on damage accumulation in irradiated α-fe, Phys. Rev. B 85 (2012) 024105. doi:10.1103/PhysRevB.85.024105.

[5] B. D. Wirth, M. J. Catural, T. D. de la Rubia, T. Khraishi, H. Zbib, Mechanical property degrada-

tion in irradiated materials: A multiscale modeling approach, Nuclear Instruments and Methods in Physics Research B (180) (2001) 23–31.

[6] C. S. Becquart, A. Barbu, J. L. Bocquet, M. J. C. et al., Modeling the long-term evolution of the primary damage in ferritic alloys using coarse-grained methods, Jnl. Nucl. Mater. 406 (2010) 39–54.

[7] R. E. Stoller, L. K. Mansur, An assessment of radiation damage models and methods, ORNL report ORNL/TM-2005/506.

[8] R. E. Stoller, S. I. Golubov, C. Domain, C. S. Becquart, Mean field rate theory and object kinetic monte carlo: A comparison of kinetic models, Jnl. Nucl. Mater. 382 (2008) 77–90.

[9] A. Bortz, M. Kalos, J. Lebowitz, A new algorithm for monte carlo simulation of ising spin systems, Journal of Computational Physics 17 (1) (1975) 10 – 18. doi:http://dx.doi.org/10.1016/0021-9991(75)90060-1.

URL http://www.sciencedirect.com/science/article/pii/0021999175900601

[10] F. Wei, J.; Kruis, Gpu-accelerated monte carlo simulation of particle coagulation based on the inverse method, Journal of Computational Physics 249. doi:10.1016/j.jcp.2013.04.030.

[11] E. M. J. M. M. K. J. Perlado, Synchronous parallel kinetic monte carlo for continuum diffusion-reaction systems, Journal of Computational Physics 227. doi:10.1016/j.jcp.2007.11.045.

[12] F. Jimnez, C. Ortiz, A gpu-based parallel object kinetic monte carlo algorithm for the evolution of defects in irradiated materials, Computational Materials Science 113 (2016) 178 – 186. doi:http://dx.doi.org/10.1016/j.commatsci.2015.11.011.

URL http://www.sciencedirect.com/science/article/pii/S0927025615007181

[13] L. J. Dooling, David J.; Broadbelt, Generic monte carlo tool for kinetic modeling, Industrial & Engineering Chemistry Research 40. doi:10.1021/ie000310q.

[14] J. Gibson, Michael A.; Bruck, Efficient exact stochastic simulation of chemical systems with many species and many channels, Journal of Physical Chemistry A 104. doi:10.1021/jp993732q.

[15] C. Domain, C. Becquart, L. Malerba, Simulation of radiation damage in fe alloys: an object kinetic monte carlo approach, Journal of Nuclear Materials 335 (1) (2004) 121 – 145. doi:https://doi.org/10.1016/j.jnucmat.2004.07.037.
URL http://www.sciencedirect.com/science/article/pii/S0022311504006385

[16] D. J. T. M. P. Allen, Computer simulation of liquids, Oxford University Press, USA, 1989.

[17] W. Mattson, B. M. Rice, Near-neighbor calculations using a modified cell-linked list method, Computer Physics Communications 119 (2) (1999) 135 – 148. doi:http://dx.doi.org/10.1016/S0010-4655(98)00203-3.
URL http://www.sciencedirect.com/science/article/pii/S0010465598002033

[18] P. M. Alexander Stepanov, Elements of Programming, 1st Edition, Addison-Wesley Professional, 2009.

[19] D. E. R. Alexander A. Stepanov, From Mathematics to Generic Programming, 1st Edition, Addison-Wesley Professional, 2014.

[20] A. Alexandrescu, C++ - Modern C++ Design. Generic Programming and Design Patterns Applied, Addison Wesley, 2001.

[21] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in c++, SIGPLAN Not. 41 (10) (2006) 291–310.

doi:10.1145/1167515.1167499.

URL http://doi.acm.org/10.1145/1167515.1167499

[22] A. C. D. G. Vlachos, An overview of spatial microscopic and accelerated kinetic monte carlo methods, Scientific Modeling and Simulation SMNS 14. doi:10.1007/s10820-006-9042-9.

[23] I. Martin-Bragado, A. Rivera, G. Valles, J. L. Gomez-Selles, M. J. Caturla, Mmonca: An object kinetic monte carlo simulator for damage irradiation evolution and defect diffusion, Computer Physics Communications 184 (12) (2013) 2703 – 2710. doi:https://doi.org/10.1016/j.cpc.2013.07.011.

URL http://www.sciencedirect.com/science/article/pii/S0010465513002397

[24] K. A. Fichthorn, W. H. Weinberg, Theoretical foundations of dynamical monte carlo simulations, The Journal of Chemical Physics 95 (2) (1991) 1090–1096. doi:10.1063/1.461138.

URL http://dx.doi.org/10.1063/1.461138

[25] P. A. Maksym, Fast monte carlo simulation of mbe growth, Semiconductor Science and Technology 3 (6) (1988) 594.

URL http://stacks.iop.org/0268-1242/3/i=6/a=014

[26] Better Code : Runtime Polymorphism, NDC Conference - 2017, http://sean-parent.stlab.cc/presentations/2017-01-18-runtime-polymorphism/2017-01-18-runtime-polymorphism.pdf, 2017.

[27] Y. Xie, D. Engler, Using redundancies to find errors, IEEE Trans. Softw. Eng. 29 (10) (2003) 915–928. doi:10.1109/TSE.2003.1237172.

URL http://dx.doi.org/10.1109/TSE.2003.1237172

[28] M. Warrier, A. Rai, R. Schneider, A time dependent model to study the effect of surface roughness

on reactivediffusive transport in porous media, Journal of Nuclear Materials 390 (2009) 203 – 206, proceedings of the 18th International Conference on Plasma-Surface Interactions in Controlled Fusion Device. doi:http://dx.doi.org/10.1016/j.jnucmat.2009.01.168.
URL http://www.sciencedirect.com/science/article/pii/S0022311509000968

[29] C. Dethloff, E. Gaganidze, V. V. Svetukhin, J. Aktaa, Modeling of helium bubble nucleation and growth in neutron irradiated boron doped (rafm) steels, Journal of Nuclear Materials 426 (13) (2012) 287 – 297. doi:https://doi.org/10.1016/j.jnucmat.2011.12.025.
URL http://www.sciencedirect.com/science/article/pii/S0022311511010695

[30] M. Leetmaa, N. V. Skorodumova, Kmclib: A general framework for lattice kinetic monte carlo (kmc) simulations, Computer Physics Communications 185 (9) (2014) 2340 – 2349. doi:http://dx.doi.org/10.1016/j.cpc.2014.04.017.
URL http://www.sciencedirect.com/science/article/pii/S0010465514001519