A NEW MULTISCALE ALGORITHM AND THE FRAMEWORK OF CODE SYSTEMS FOR ADVANCED DEFECT CLUSTER DYNAMICS SIMULATIONS

A Dissertation

by

JIANGYUAN FAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Lin Shao
Committee Members,	Pavel V. Tsvetkov
	Sean M. McDeavitt
	Ibrahim Karaman
Head of Department,	John E. Hurtado

May 2019

Major Subject: Nuclear Engineering

ABSTRACT

In the research of void swelling for reactor material, the dynamics of defect clusters is an important topic during the research of principles for this phenomenon. The simulation of defect cluster evolution pursues more atomic level details on cluster dynamics and higher calculation capability to evolve the system to higher scale at the same time, which is a great challenge to nuclear engineering. For this study, a new multiscale algorithm combining the concept of first-passage process, asynchronous event-driven system and separate solvers design has been developed to provide a new solution to the simulation of defect cluster diffusion and reaction system. This project covers the design of algorithm, the design of framework of code systems and the development of prototype code. The new method is able to simulate the full dynamics of defect cluster reaction and diffusion. It greatly lowers down the time complexity in tradition kinetic Monte Carlo simulation systems, accelerates the simulation for large scale, overcomes the troubles to handle local events and guarantees a flexibility to simulate multiple scenarios within same framework.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Shao, and my committee members, Dr. Tsvetkov, Dr. McDeavitt and Dr. Karaman, for their guidance and support in this research. A special thanks to Dr. Shao, my committee chair, for his countless hours throughout this research of encouraging and helping with the topic.

Thanks to my friends, colleagues, the faculty and the staff in Texas A&M University for making my years here a precious period of my life. Special thanks to my friends Ran Lu and Jiani Hu for their selfless help on the preparation of the defense and advices on the presentation.

Finally, I would like to thank and dedicate the work to my parents, Fuping Fan and Xinwei Jiang, for their encourgement to my research life and supporting to my daily life.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a dissertation committee consisting of Professor Lin Shao, Pavel V. Tsvetkov and Sean M. McDeavitt of the Department of Nuclear Engineering and Professor Ibrahim Karaman of the Department of Materials Science and Engineering.

All the work conducted for the dissertation were completed by the student independently.

Funding Sources

This work was made possible in part by the US Department of Energy, Office of Basic Energy Sciences, under Grant No. DE-SC0006725, and the US National Science Foundation, under Grant No. 1708788. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the US Department of Energy, Office of Basic Energy Sciences or the US National Science Foundation.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

Page

LIST OF TABLES

1. INTRODUCTION

As one of important phenomena in the study of radiation damage to material in nuclear reactors, swelling has been widely researched since last century. This phenomenon, which arises with changes both on mechanic features and thermodynamic features, challenges the application of different alloys in reactor design both with safety concerns and efficiency concerns. Through a long period of time, numerous researches have been processed and also proceeding to understand the theory behind [1] [2] .

1.1. Background

During last decades, several important breakthroughs on the microstructure researches revealed part of the facts on the mechanism for swelling [2] .

Experimental data were collected to understand the changes on microstructure due to irradiation under different conditions [4] [5] [6] . It has been revealed that structure defects would be introduced to existing structure during irradiation, which consists of vacancies and interstitials. A mechanism, which consists of diffusion and reaction (coalescence, annihilation and dissociation), takes control of the evolution of defects and defect clusters and results to the buildup of large clusters which are known as voids [1] [2] , which has been regarded as the key factor of swelling.

Back to 1970s, numberous researches has already shown the relationships between α-iron and ferreric ferritic steels' swelling characteristics and different irradiations [2] , which includes elevated temperature electron irradiation [7] [8] [9] , ion irradiation [10] [11] [12] and limted neutron irradiation [13] [14] .

As shown in Fig. 1.1, experimental data shows the growth of voids as white space under transmission electron microscope (TEM) and will be affected by irradiation conditions, such as the buildup speed of radiation damage which is known as displacement per atom (DPA) rate.

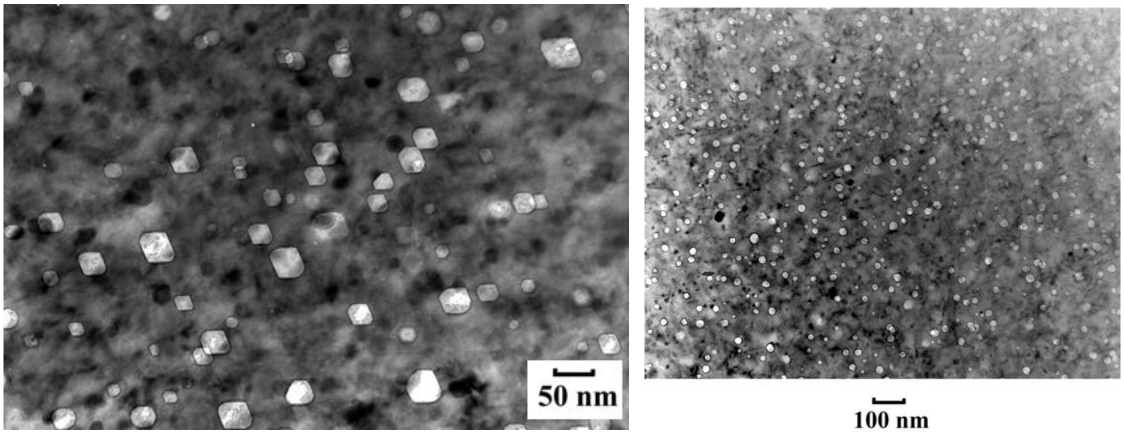


Figure 1.1 Demonstration of the Effect of DPA Rate on Swelling of Annealed AISI 304 Stainless Steel in EBR-II, reprinted from [4] [5] .

In addition to these microscopic facts, researchers keep searching for a full-range picture to the evolution and a valid method to simulate the process. For this topic, the problem has been promoted to cover all different scales in time and space, which requires the calculation and its systems with certain multiscale capabilities.

Multiscale simulation requires a system covering a wide range of scale both on time and space, a typical system for the research of mechanical properties of metals [15] is described in Fig. 1.2. Generally, the whole system is separated into three different scale: microscale, mesoscale and macroscale, where microscale researches focus on the atomic level facts, mesoscale researches focus on dynamics and evolution and macroscale researches focus on phenomena and characteristics with experimental data.

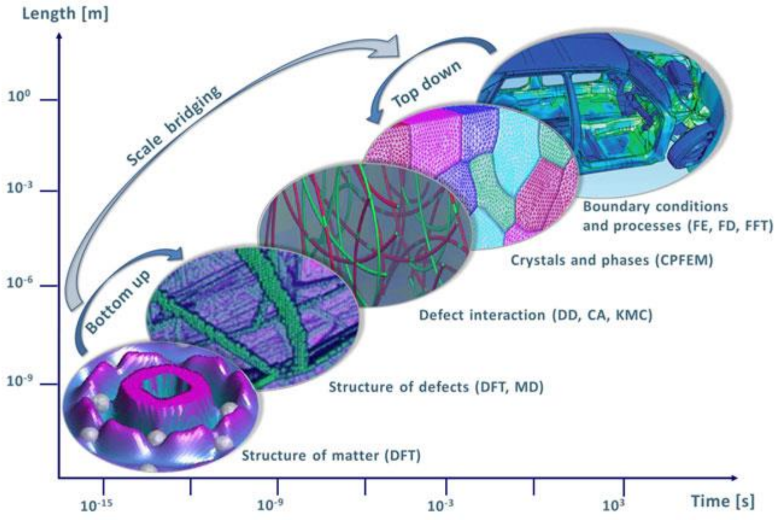

Length [m]
10⁰
10⁻³
10⁻⁶
10⁻⁹
Scale bridging
Top down
Bottom up
Boundary conditions
and processes (FE, FD, FFT)
Crystals and phases (CPFEM)
Defect interaction (DD, CA, KMC)
Structure of defects (DFT, MD)
Structure of matter (DFT)
10⁻¹⁵
10⁻⁹
10⁻³
10³
Time [s]

Figure 1.2 Examples of the Different Time and Space Scales Encountered in the Field of the Metal Mechanical Properties, reprinted from [15] .

Typical multiscale systems require the combination of multiple algorithm with extra bridging systems [15] , extra approximations for the sub-systems are required to support the extra bridging design. In order to limit the loss on accuracy and improve the system efficiency, the extension of capability on each scale is critical.

In addition to this general requirement, a challenge for defect cluster dynamics simulation arises due to limitations on current mesoscale algorithm, mean-field rate theory (MFRT) and kinetic Monte Carlo (KMC) method. This results to a situation that no solution is provided to simulate the full range cluster dynamics in this area.

1.2. Motivations to the New Solution

For the simulations applied to describe atomic level evolutions, Monte Carlo (MC) method is always a good example to introduce. It is a typical method of solving the Schrödinger equation to simulate the ramdon diffusion [16] . In general MC method, a particle is regarded to be transported to its neighbor positions with certain possibilities, with a sampling of random seed, the system decides to which direction the particle will be transported. It can be figured that this method concerns too much on these atomic level hops which prohibits the algorithm to reach a large scale calculation limited by calculation capability.

A faster way to process the evolution is to apply KMC method [20] . KMC method considers all possible location changes to all movable particles. Based on the possibilities to change from one distribution to different distributions, a "table of possibilities to change status" is built for sampling. For KMC method, the sampling is no long focusing on each particle, instead, it focus on the whole system and makes larger hops comparing to the ones in MC method. This gives one good solution on simulation work to the evolution problem in mesoscale and has produced several great breakthroughs [85] .

The KMC method still considers the system in atomic level. It has been widely applied to systems which has a mechanism to evolve by reactions among random walkers, including diffusion-controlled chemical and biochemical reactions, recombination of carriers in semi-conductors, Ostwald ripening and cluster dynamics [17] [19] . This method has a clear description to the dynamics and enough atomic level details. For diffusion research, it considers all possible hops for every movable member in the system (noted as walker), calculates the hop rate for each walker and use a random seed to choose a certain hop event to take place as next event to change the system status. At the same time, another random seed is introduced to update the system clock [18] .

As described in the last section, challenges for the simulation arise due to the limitations in current mesoscale algorithm. KMC method is limited by its requirements on calculation capability. Research shows for a simulation in relatively realistic event number, the required CPU hours rises to tens of years even with simple dynamics [20] [21] . This feature limits lots of the application with KMC method in microscale. Adding to this, the application of KMC method has a challenge to describe the dynamics in a system with multiple types of walkers, and results to the inflexibility on the code systems.

Meanwhile, the MFRT method, one of the widely used algorithms, considers the system with object concentrations, uses change rates to define the material system and irradiation conditions [1] [22] . The master equation of MFRT is written as:

$$\dot{C}_i = g_i + \sum_j s_{ij} C_i + \sum_k d_{ik} C_i C_k \tag{1.1}$$

where C_i refers to the concentration of *i* type object; g, s, d defines the reaction rates for the object in different reaction types.

MFRT method was firstly developed to void growth [23] . This approach was further-developed [24] [25] during 1970s for this area. The simulation of surface reaction together with diffusion-coalesence system was then introduced [26] [27] . Further development has been introduced related to sink strength [28] [29] [30] [31] [32] , multiple sink effects [33] , gas effects [34] [35] , spatial arrangement and defects generation rate [36] [37] [38] [39] [40] , temperature shift effect with dose rate [41] , free surface effects [42] [43] [44] [45] [46] [47] , ion-injection during irradiation [48] [49] , solute segregation effects [50] [51] [52] [53] [54] , and impurity trapping effects [55] [56] [57] [58] [59] . Further research also linked this approach to void nucleation [60] , creep and stress affected swelling [61] [62] and other properties' changes [63] . This approach has been accepted and widely used in radiation damage study since 1960s [64] [65] [66] [67] [68] . However, for realistic high dose research, the number of equations raises to unacceptable number and causes the combinatorial explosion [22] . Simplifications as ignore complex species are required for realistic research, which also brings inaccuracy. Another challenge for MFRT comes from the dimensionless feature of this method, as the mean-field assumption cancelled the spatial difference on objects, MFRT is not valid to describe diffusion with sufficient atomic level details.

Addition to KMC method and MFRT method, multiple other solutions have been proposed with different research interests [69] [70] [71] [72] . With a focus on several different features in related problem, these solutions offers adventures to specific regions with a disadvantages to others. It is challenged to apply any of these solutions by the case sensitivity and algorithm limits to the research of defect cluster dynamics and related buildup of voids.

Challenged by the features mentioned above, the multiscale simulation for defect cluster dynamics requires a new algorithm and code systems for its mesoscale evolution. It is designed to overcome the limitations from both KMC and MFRT. Comparing with KMC, it requires a better calculation performance which is able to reach high scales much faster, meanwhile, with similar accuracy as KMC. Comparing with MFRT, it requires more details on atomic level behavior with high calculation capability and less simplifications. The new algorithm is required to have the capability to link both microscale events and macroscale characteristics. For the code systems, flexibility is in need to extend the application to different phenomena.

Based on these requirements, the First-Passage Kinetic Monte Carlo (FPKMC) [21] algorithm and the concept of first-passage method [73] is taking into the consideration as parts of the basic design concepts. This method applies the first-passage method which will be introduced in next section and uses Green's function [21] [73] to apply analytic solutions to Monte Carlo sampling.

The FPKMC algorithm considers to handle the whole dynamics system with N 1-body or 2-body problems instead of an N-body problem with a design of event-driven asynchronous system [21] . The procedure is described as: 1) drawing non-overlapping protective domains (PD) for all walkers; 2) sampling times of first arrival with Green's function for each walker, registering in a queue; 3) finding the shortest time of first arrival; 4) moving the respective walker to a randomly selected point on its PD; 5) constructing a new PD for the particle which changed position; 6) sampling new event times only for the particle moved, inserting in the queue; 7) back to step 3. This asynchronous process is proved to decrease the time complexity from O (N^2) to O (1) comparing to synchronous Green's function method [74] , meanwhile, the Green's function First-Passage method greatly shorten the calculation time without a loss of accuracy.

Researches with similar concepts have shown great advantages as: up to 6 orders of magnitude faster comparing to Brownian Dynamics [75] , speed up from tens of years CPU hours [21] [81] [82] to 36 CPU hours comparing to traditional KMC code [21] , same accuracy [21] [75] and statistic behavior [78] . With the huge advantage on the behavior, this method has been accepted and developed both in biochemistry [75] [76] [83] [84] and material science [21] [78] [79] from the original single walker research [86] [87] and ground state of boson research [74] [86] [88] to recent advanced research[21] [83] [84] combining with different features as volume replication [21] [89] to further advance the advantages.

However, there is a critical challenge once first-passage method is considered in the defect cluster dynamics in radiation research. Currently, first-passage method is limited by local *N*-body case [77] [78] [79] . For realistic case in radiation research, a compacted space cannot be avoid in the damage cascade scenario and high dose system. As *N*-body system cannot be solved analytical, the protective domain design in current system is not able to be maintained.

Several attempts or possible solution is under research, as time-driven Brownian Dynamics (BD) / Molecular Dynamics (MD) hybrid [75] [77] [79] [80] . Limited by MD definition and the huge calculation requirement, these solution limits the flexibility and also the event-driven advantages.

Based on the concept of first-passage method and the FPKMC algorithm and the research requirement of defect cluster dynamics, a new solution is given to overcome the limits from current first-passage methods. In this solution, an event-driven modular framework is built, and a mixed core is introduced. Named as advanced defect Cluster Reaction Dynamics (aCRD), this project offers a new solution on algorithm and code systems design, and also comes with a prototype to initial the research to understand the multiscale evolution for void swelling.

The following sections in this dissertation will introduce the concept of project aCRD and the design of algorithm, code systems and prototype code in details.

2. PROJECT OVERVIEW

Generally, aCRD is an algorithm with KMC core, following with first-passage method and organized with separate solvers design. For this research is interested in a dynamics with both diffusion and reaction, the aCRD separates these two questions and rearrange the concept from general KMC logic. In this section, these concepts will be introduced with more details.

2.1. Green's Function First-Passage Process

With the same approach to treat the diffusion equation as FPKMC, the aCRD algorithm relies on the first-passage and non-passage propagators to evolve walkers (defect clusters) during diffusion. It comes with the concept of first-passage process and the function related to the solving of Green's function.

2.1.1. First-Passage Process

The *first-passage probability* refers to the probability that a diffusing particle or a random-walk first reaches a specified site at a specified time [73] . With such a probability to reach a certain status, the first-passage process treats the transportation in diffusion style of defects to a certain boundary [73] .

As shown in Fig. 2.1, at point A, the particle exists the domain at its first time, which triggers the first-passage event. Once the space is well assigned to each domain to cover only 1 defect, the problem of defect diffusion inside its domain is able to be described with its analytical solution. Also, in this way, numerous hops for the defect inside its domain Ω before the existing of boundary which has to be concerned in traditional KMC diffusion are not required to be concerned any more.

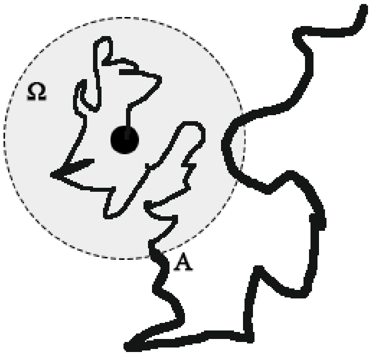


Figure 2.1 One Particle in Random Diffusion, the Particle Diffuses in Two Stages: Before First Existing Ω (Fine Trajectory) and After (Bold).

With first-passage process, the efficiency of calculation will be greatly improved, especially for the case with large domains [78] .

2.1.2. Green's Function Propagators and Example of Sampling

In this section, the Green's function sampling method is introduced with an example of general diffusion following general diffusion equation in a continuum-space continuum-time random walk. Several functions are required as parameters refer from FPKMC [78] .

Following general diffusion equation as:

$$D\Delta c(\mathbf{r},t) = \partial c(\mathbf{r},t)/\partial t \qquad (2.1)$$

where D is the diffusivity of respective defect, $c(\mathbf{r},t)$ is the probability density of finding the diffusing defect in an infinitesimal volume around **r** at time t, also can be regarded as a similar term to concentration. The diffusion follows an initial condition $c(\mathbf{r},0) = \delta(\mathbf{r}-\mathbf{r}_0)$ and boundary condition $c(\partial\Omega ,t) = 0$.

With a domain set up as shown in Fig. 2.1, the probability that the walker has not crossed the boundary $\partial\Omega$ of the domain Ω at time t from its original $t = 0$, also defined as survival probability S(t), is:

$$S(t) = \int_\Omega c(\mathbf{r},t)d\mathbf{r} = 1\text{-}D \int_0^t \int_{\partial\Omega} \nabla c(\mathbf{r},\tau)\cdot\hat{n}|dA|d\tau \tag{2.2}$$

where $c(\mathbf{r},t)$ is the Green's function of the diffusion equation in its domain Ω.

In order to sampling the final status of first-passage process, the exiting time and location are required. These information link to other two parameters: the exiting probability per unit time p(t) and the splitting probability $j(\mathbf{r},t)$ defined as the probability density for the exit location on boundary surface ($\mathbf{r}\in\partial\Omega$). These two functions can be described as:

$$p(t) = -\partial S(t)/\partial t \tag{2.3}$$

and

$$j(\mathbf{r},t) = \frac{\nabla c(\mathbf{r},t)\cdot\widehat{n_\mathbf{r}}}{\int_{\partial\Omega}\nabla c(\mathbf{r}',t)\cdot\widehat{n_\mathbf{r}}dA} = D\frac{\nabla c(\mathbf{r},t)\cdot\widehat{n_\mathbf{r}}}{-p(t)},\ \mathbf{r}\in\partial\Omega \tag{2.4}$$

where $\widehat{n_\mathbf{r}}$ is the outward normal direction vector to the boundary surface $\partial\Omega$.

At the same time, function $g(\mathbf{r},t)$ is defined as probability density to find the particles near $\mathbf{r}$ at time t under the condition that it has not exited the domain by time t:

$$g(\mathbf{r},t) = c(\mathbf{r},t)/S(t) \tag{2.5}$$

This function is design for the sample of walker location inside its domain at a specific time *t*.

In order to do sampling with these functions, c($\mathbf{r}$,t), the Green's function to the original diffusion equation is required to initial the calculation. Here, an example of solution is list as an example from the solving of one dimension diffusion on [0, 1] following:

$$\partial^2 c/\partial x^2 = \partial c/\partial t \tag{2.6}$$

$$c(0,t) = c(1,t) = 0,\ \ c(x,0) = \delta(x\text{-}x_0)$$

The diffusion equation 2.6 has its analytic solution to be written as the Eigen-function expansion:

$$c(x,t) = 2\sum_{k=1}^{\infty} \sin(k\pi x) \sin k\pi x_0\, e^{-k^2\pi^2 t} \tag{2.7}$$

This expansion is able to be converged on $t \geq 1/\pi^2$ and able to be solved on $t \leq 1/4$ [18].

By using smooth connection function between both ranges, a full range solution is obtained for sampling.

An example of Green's function sampling in single-sphere system is provided in other research [78] [79] . An introduction to the concept is introduced here, together with the design applied in aCRD prototype solver with related approximation.

For the case in 3D system of spherical clusters protected in spherical HPD, two series expansions of the solution converging at different time region is described in the following equations Eq. 2.8 and Eq. 2.9:

$$c(r,t) = (4\pi t)^{-1.5} \sum_{m=-\infty}^{\infty} (1+2m/r) \exp[-(r+2m)^2/4t] \tag{2.8}$$

which converges quickly on t < 1/4 in the unit of L^2/D where L is the HPD radius and D is the diffusivity to this walker;

$$c(r,t) = (1/2r) \sum_{m=1}^{\infty} m \sin(m\pi r)\, e^{-m^2\pi^2 t} \tag{2.9}$$

which converges quickly at long times ($t > 1/\pi^2$).

Based on the definition, Eq. 2.8 and Eq. 2.9 are regarded to be the solutions (Green's function) to related diffusion equations. Integration of $c(r,t)$ over the unit sphere described as the related HPD offers the survival probability S(t). From previous research [79] , related functions are changed into two infinite series forms:

for Eq. 2.8:

$$S(t) = S_{n\to\infty}(t) \tag{2.10}$$

where

$$S_n = (2\pi t^3)^{-\frac{1}{2}} \sum_{m=-n-1}^{n} \left[2m+\frac{(1+2m)^2}{2t}\right] \exp\left[-\frac{(1+2m)^2}{4t}\right]$$

and for Eq. 2.9:

$$S(t) = -2\pi^2 \sum_{m=1}^{\infty} (-1)^m m^2 e^{-m^2\pi^2 t}. \tag{2.11}$$

Eq. 10 and Eq. 11 are designed to use piecewise smooth function C(t) to connect each series with a switchover time $1/4 \geq \tau \geq 1/\pi^2$. In this way, the exit time *t* is able to be calculated by solving $C(t) = \xi$, where ξ is a random number uniformly distributed in [0,1). In previous research, $\tau = 0.243$ is selected to perform a rejection ratio of about 0.6%. Based on the same concept to design another over-estimator, E02 event series is also able to be sampled.

However, the prototype solver applied to obtain verification data to current aCRD code systems will only sample the data by solving the part on $t > \tau$, which makes $C(t) \to S_1(t) = 2\pi^2 e^{-\pi^2 t}$. This method results in the increasing of rejection ratio and maximum relative error in survival probability, and may require further update to modify before any validation work to aCRD system.

2.2. Separate Solvers Concept

One of the most important features in aCRD algorithm, as mentioned in early section, is the separation of events in different spatial range with different solvers. This design is given to overcome the limits on local *N*-body problem [78] [79] in current first-passage methods. In this solution, an event-driven modular framework is built, and a mixing core with both FPKMC and other algorithm (e.g. MFRT) is introduced. In this way, the *N*-body diffusion and reaction problem is separated into two parts and handled with different protective domains. In aCRD, these PDs are noted as hard protective domains (Hard PD, HPD) and soft protective domains (Soft PD, SPD). This givens the overall concept of the advanced Cluster Reaction Dynamics (PROJECT aCRD).

Fig. 2.2 shows an overall scheme comparing general KMC algorithm, FPKMC and PROJECT aCRD.

Following with Green's function first-passage method diffusion, in aCRD, while the walkers (defect clusters) are transported into a space where the consideration of *N*-body problem is necessary, the walkers are designed to switch from FPKMC algorithm controlled space (HPDs as described) to event-driven local reaction space (SPDs as described) which is controlled by other algorithm as rate theory. This design separates the analytically solvable 1-body problem and unsolvable *N*-body problem, while an event-driven *N*-body solver is designed to solve the *N*-body reaction problem separately.

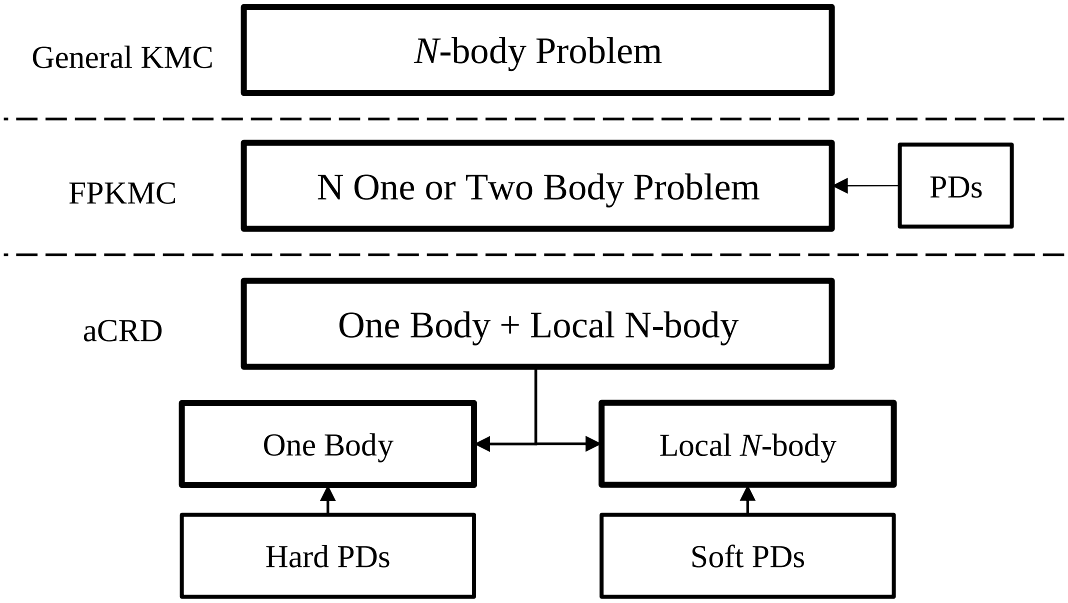


Figure 2.2 Comparison of KMC/FPKMC/aCRD and aCRD Scheme.

In the local space, PROJECT aCRD offers an interface for switching between different local *N*-body solvers, to guarantee flexibility for different purposes. In the design of aCRD, it is capable with follow features:

- compatible with most rate theory solvers (e.g. MFRT, stochastic cluster dynamics)
- able to simulate multi-dimension diffusion at the same time
- able to use SPDs to simulate special structures (e.g. Grain Boundaries)
- compatible with MD/BD solvers (KMC/MD mixed, hybrid-driven design)
- reach multiscale events within one framework by shifting algorithm

2.3. Algorithm and Code Systems Scheme

This section describes the basic structure to PROJECT aCRD by offering schemes which gives an overview to different modules and their connection modules. Details on the design of algorithm and code systems is included in Section 3 while the details of prototype are included in Section 4.

2.3.1. Algorithm Structure

As the content in Section 2.2., aCRD algorithm consists of two different events and different space to support the event flow. The overall schemes is given in Fig. 2.3.

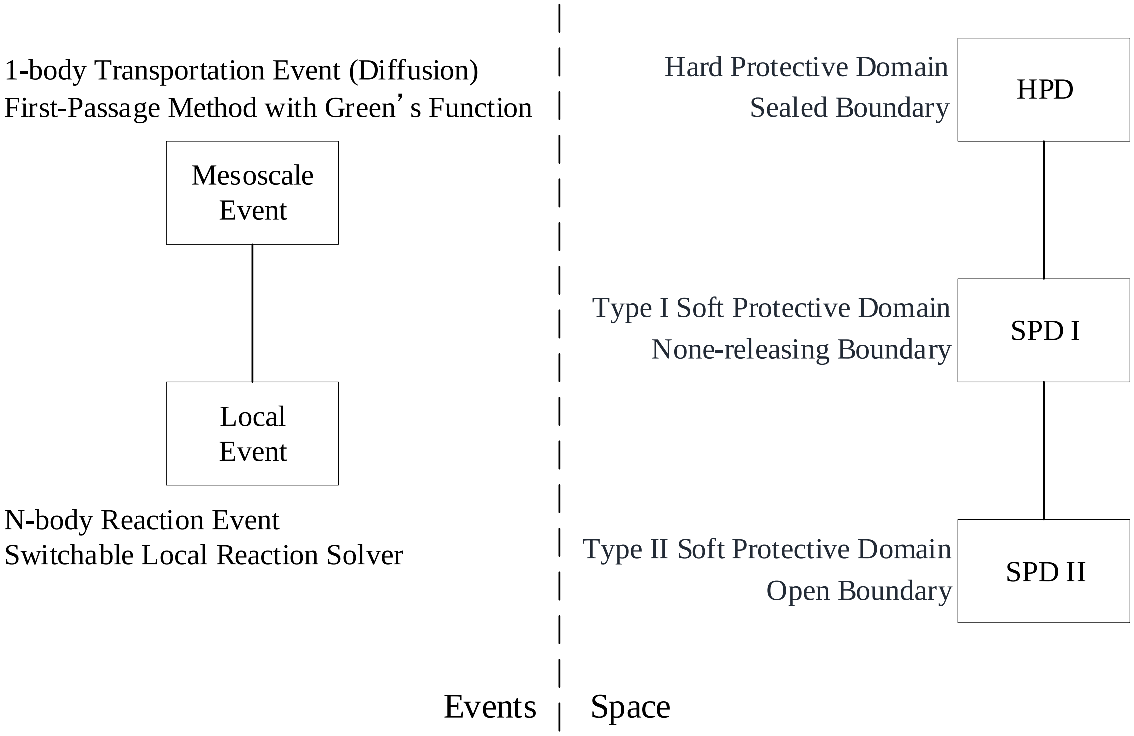

1-body Transportation Event (Diffusion)
First-Passage Method with Green's Function
Mesoscale Event
Local Event
N-body Reaction Event
Switchable Local Reaction Solver
Hard Protective Domain
Sealed Boundary
HPD
Type I Soft Protective Domain
None-releasing Boundary
SPD I
Type II Soft Protective Domain
Open Boundary
SPD II
Events
Space

Figure 2.3 aCRD Algorithm Scheme.

In aCRD, the events consist of mesoscale event and local event, which are described with different mechanism. The mesoscale event is designed to handle 1-body transportation event. Green’s function is able to be applied for this type of event to obtain analytic solution. The local event is designed to serve local *N*-body reaction event. Switchable local solver is designed to process a rate theory based calculation and to sample the final status.

Three types of space are designed to support the event flow, which is hard protective domain (HPD), type I soft protective domain (SPD I) and type II soft protective domain (SPD II).

HPD is designed to be an event-driven domain with sealed boundary space to handle 1-body diffusion event. Every HPD contains 1 walker, any walker cannot cross the domain boundary without breaking the domain. This design secures that any walker in HPD can be regard as 1-body transportation and maintain the accuracy from Green’s function solver.

SPD I is designed to be an event-driven domain with non-releasing boundary to handle local *N*-body event. SPD I contains multiple walkers, walkers can enter the boundary without breaking the domain, but cannot do the opposite. In this way, SPD I serves as a fixable space to handle local reactions without lower down the event-driven advantages.

SPD II is designed to be a hybrid-driven domain with open boundary to handle special structure definition or time-driven solver for different purpose.

2.3.2. Code Systems Design

Adding to the algorithm described in Section 2.3.1, code systems are built to support the design. This gives another half of PROJECT aCRD. The overall schemes of code systems is given in Fig. 2.4.

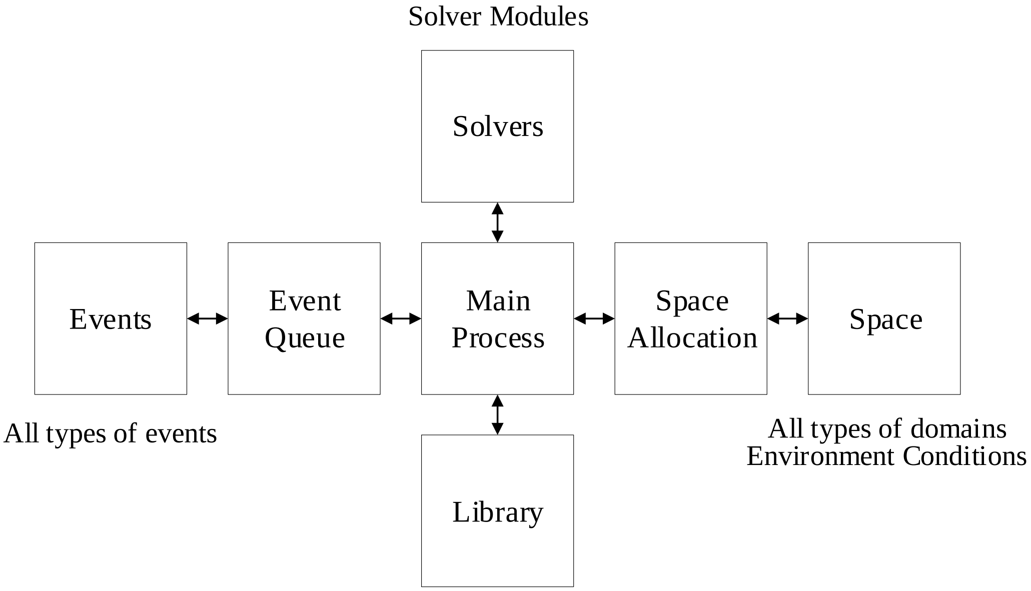


Figure 2.4 aCRD Code Systems Scheme.

The design of code system is following the concept of modules to separate different parts into separate modules in order to guarantee a handy procedure during update.

In this design, separate solver modules maximizes the flexibility for future upgrades both on enabling new features and improving of accuracy. Library module is also independent from other parts which offers interfaces for main process to read parameters from database. At the same time, the space allocation is able to be independent from the main procedure, this guarantees the flexibility to define the simulation box with more details. In the design, the event flow from generation of events to execution of events is also able to be independent, this makes it handy to describe new events (mechanism) to describe different system.

3. ADVANCED DEFECT CLUSTER REACTION DYNAMICS

The section describes the details on the design of PROJECT aCRD algorithm and the code systems. As described in the last section, aCRD treats the diffusion and reaction dynamics in an event-driven asynchronous style. A scheme of dynamics is described in Fig. 3.1.

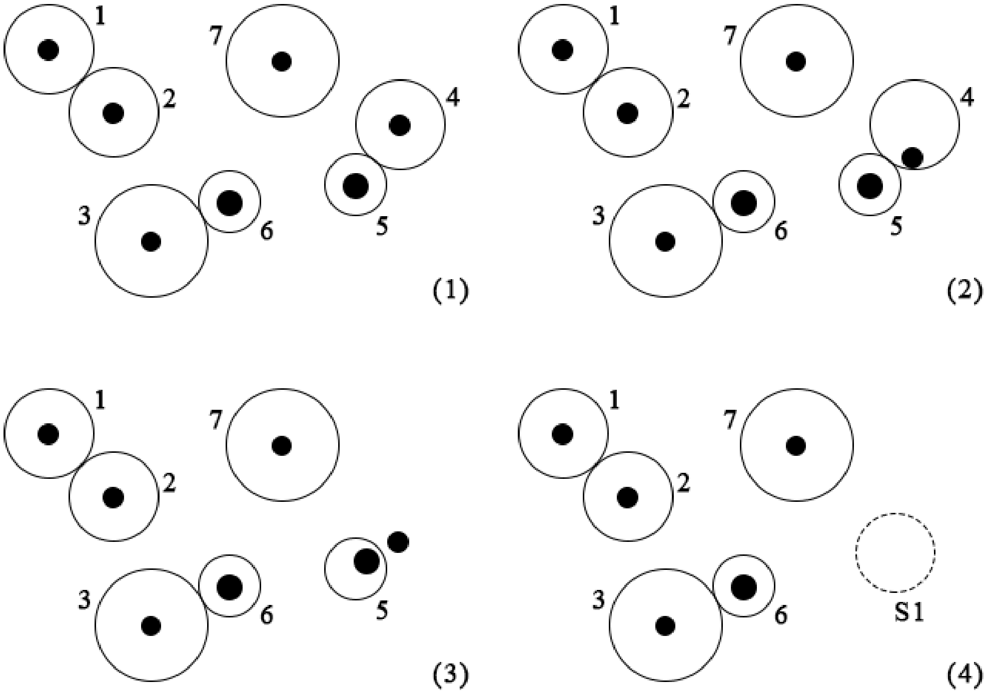


Figure 3.1 Example of aCRD Diffusion and Reaction Dynamics Scheme.

As described in Fig. 3.1, different diffusing defect clusters (black dots) are located in the simulation space and protected by non-overlapping HPDs numbered from 1 to 7 for each defect. All these defects have different clock by themselves, and will be triggered with a diffusion event to the inner boundary of its HPD at certain time to a certain location which is sampled from the Green's function solver. From Fig. 3.1 (1) to Fig. 3.1 (2), the defect protected in HPD-4 is regarded to be the first defect to trigger its diffusion event and be transported to the edge of the domain. After the transportation, the diffused defect is supposed to be protected again in a new non-overlapping HPD and enter the event list again, then process to next event.

However, once the case shown in Fig. 3.1 (2) is triggered while the transported defect is too close to another HPD, which is HPD-5 in this case, its neighbor will go through a transient diffusion event which will synchronize the defect in HPD-5 to a random sampled location inside its HPD (refer to Eq. 2.5 in Section 2.1.2), as shown from Fig. 3.1 (2) to Fig. 3.1 (3). Generally, after this event, both defects which have been transported by a diffusion event and another transient diffusion event will be protected to new HPDs respectively and enter the event list waiting for next event in list.

The collision, or reaction, takes place in the case once the transported defects are too closed to another, as shown in Fig. 3.1 (3). This will trigger the buildup of a SPD to protect both defects in a single domain as shown in Fig. 3.1 (4) numbered as S1 with boundary in dash line. As the SPDs allow new defect to enter its boundary, a new HPD is allowed to overlap a SPD which is the entry of new defect to the SPD, but is not allowed to fully cover a SPD. Once the event in a SPD is triggered, the local solver will be required to sample the final status to the SPD's current members. New defects are released with same clock to the space inside the original SPD. These defects will follow general rules to be protected again and enter the event list after the break of the SPD.

The steps above give the outline of aCRD event logic, the algorithm and code systems are designed to maintain this logic.

3.1. Algorithm

Comparing to a time-driven or hybrid-driven system, a fully event-driven system is advanced on its original time complexity from O (N^2) to O (1). At the same time, an event-driven system avoids the inflexibility from the accurate MD definition requirement from *N*-body time-driven MD algorithm, and also the possible holdup from the local calculation. In this way, the event-driven design is able to maximize the calculation advantages and enable the possibility of larger scale simulation both on time scale and also space scale.

In order to keep the event-driven description through this research, a new event-driven dynamics to handle the defect clusters diffusion and reaction is required. This new event-driven defect cluster dynamics is required to describe all defect behaviors in the style of events and to combine all types of events in one system. In the other word, this dynamics offers a physical background to the algorithm and becomes one the most important contents in this project.

This section describes all the details on the setup of this dynamics network and a multi-stage algorithm designed to optimize the logic, which is named as area synchronize multi-stage (ASMS) procedure.

3.1.1. Dynamics and Algorithm

In this original design in aCRD algorithm, the event-driven defect cluster dynamics consists of five major types of events and multiple sub-events. These events form the network of dynamics and profile the diffusion and reaction system into the style of multiple single events. This section describes the whole algorithm in the form of different type of events and the layout of the original flowchart in the form of events.

3.1.1.1. Events Network of Dynamics

In the design of aCRD algorithm, the event-driven defect cluster dynamics consists of five major types of events and multiple sub-events. This section describes the whole algorithm in the form of different type of events. In all, the events network of dynamics consists:

- Major Diffusion Event (Type 01 event, E01)
  - Main event
  - Type 1 conjunction event
  - Type 2 conjunction event
- Transient Diffusion Event (Type 02 event, E02)
  - Main event
  - Type 1 conjunction event
  - Type 2 conjunction event
- Reaction Event (Type 10 event, E10)
  - Main Event
  - Conjunction events
- Dissociation Event (Type 11 event, E11)
  - Main event
  - Conjunction events

- Irradiation Event (Type 20 event, E20)
    - Main event

Major diffusion event (E01) series describe the dynamics for a defect cluster to migrate inside a HPD from its original location to the boundary by diffusion and link the transportation to other types of events. Transient diffusion event (E02) series describe the dynamics for a defect cluster to migrate inside a HPD from its original to a random location inside the domain and link the hop to other types of events. These two series form the framework with the application of Green’s function and the design with HPD.

Reaction event (E10) series describe the dynamics for several defect clusters to form SPD and to shift the algorithm from first-passage diffusion to other algorithm. This series concerns the links between different algorithms.

Dissociation event (E11) series and irradiation event (E20) series are both functional events designed to describe the dynamics with more details and to fit with different scenario. They concerns with several subroutines to add details and new events to a general diffusion and reaction system.

3.1.1.1.1. Major Diffusion Event (E01) Series

The main event of major diffusion (type 01 event series) is described as Fig. 3.2, where the black dot represents a defect cluster while the closed circle is its HPD.

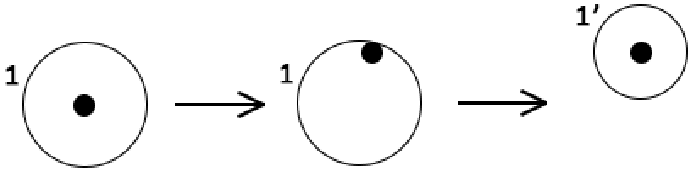


Figure 3.2 aCRD E01 Main Event Scheme.

This sub-event links one major diffusion event (E01) to a random event in event queue. This sub-event results in the break of one existing HPD and the buildup of one new HPD. To the defect cluster, this sub-event turns the cluster from HPD protected status to a released status, finished with the turn to the HPD protected status by another HPD. In the style of algorithm, this event is described as:

1) Drawing non-overlapping HPDs for all *N* walkers;

2) Running the Green's function solver, sampling first arrival time, registering them in the global event queue;

3) For the walker related to the first event in queue, sending it to its respective final location and breaking the HPD;

4) Checking location, not too close to another HPD or inside of a SPD;

5) Rebuilding a non-overlapping HPD for the transported walker;

6) Rerunning the Green's function solver, sampling time of arrival for the newly protected walker, updating the event queue, processing to next event.

In this event, the Green's function solver is required, also with several location checks to avoid the execution of conjunction events.

There are two types of conjunction events in type 01 event series. They are designed to link this event series to other event series.

The type 1 conjunction event of major diffusion (type 01 event series) is described as Fig. 3.3, where the black dots represent defect clusters while the closed circles are their HPDs. In this sub-event, several defect clusters and their HPDs are concerned.

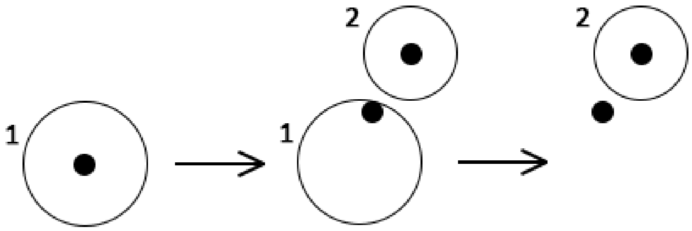


Figure 3.3 aCRD E01 Type 1 Conjunction Event Scheme.

This sub-event links one major diffusion event (E01) to one transient diffusion event (E02). This sub-event results in the break of one existing HPD without the rebuild. To the defect clusters, this sub-event turns one cluster from HPD protected status to a released status. In the style of algorithm, this event is described as:

1) Drawing non-overlapping HPDs for all *N* walkers;

2) Running the Green's function solver, sampling first arrival time, registering them in the global event queue;

3) For the walker related to the first event in queue, sending it to its respective final location and breaking the HPD;

4) Checking location, (*IF*) too close to another HPD;

5) (*THEN*) go to transient diffusion (type 02 event series).

In this event, the Green's function solver is required, also with several location checks to initial the link to other event series.

Adding to this type of conjunction events, the type 2 conjunction event of major diffusion (type 01 event series) is described as Fig. 3.4, where the black dot represents the defect cluster while the closed circle is its HPD. The circle with dash boundary represents a SPD overlapped with the original HPD. In this sub-event, one defect cluster and its HPD are concerned, together with a SPD.

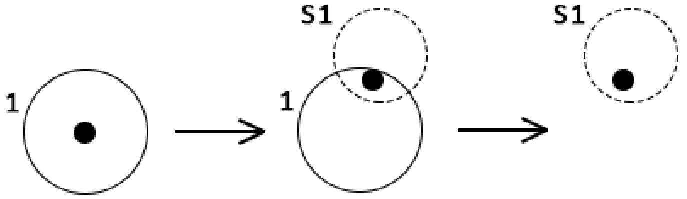


Figure 3.4 aCRD E01 Type 2 Conjunction Event Scheme.

This sub-event links one major diffusion event (E01) to one reaction event (E10). This sub-event results in the break of one existing HPD without the rebuild. To the defect clusters, this sub-event turns one cluster from HPD protected status to a released status. In the style of algorithm, this event is described as:

1) Drawing non-overlapping HPDs for all N walkers;

2) Running the Green's function solver, sampling first arrival time, registering them in the global event queue;

3) For the walker related to the first event in queue, sending it to its respective final location and breaking the HPD;

4) Checking location, (*IF*) inside of a SPD;

5) (*THEN*) go to reaction event (type 10 event series).

In this event, the Green's function solver is required, also with several location checks to initial the link to other event series.

Three sub-events described in this section form the part of network initialized from the break of HPD and forms the E01 major diffusion event series.

3.1.1.1.2. Transient Diffusion Event (E02) Series

The main event of transient diffusion (type 02 event series) is described as Fig. 3.5, where the black dots represent defect clusters while the closed circles are their HPDs.

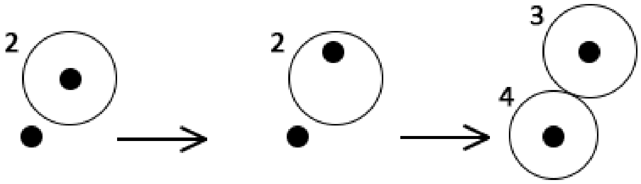


Figure 3.5 aCRD E02 Main Event Scheme.

This sub-event links one transient diffusion event (E02) to a random event in event queue. This sub-event results in the break of one existing HPD and the buildup of several new HPDs. To the defect cluster, this sub-event turns the cluster from HPD protected status to a released status, finished with the turn of all released clusters to the HPD protected status by other HPDs. In the style of algorithm, this event is described as:

1) Running Green's function solver for the neighboring protected walker and sending it to the new location inside original HPD (Local Synchronization);

2) Breaking the respective HPD;

3) Checking location, not VERY close or inside of a SPD;

4) Rebuilding non-overlapping HPDs for both transported walkers;

5) Rerunning the Green's function solver, sampling time of arrival of the newly protected walker, updating the event queue, processing to next event.

In this event, the Green's function solver is required, also with several location checks to avoid the execution of conjunction events.

There are two types of conjunction events in type 02 event series. They are designed to link this event series to other event series.

The type 1 conjunction event of transient diffusion (type 02 event series) is described as Fig. 3.6, where the black dots represent defect clusters while the closed circles are their HPDs. The circled with dash boundary represents a SPD contains multiple clusters. In this sub-event, one defect cluster and its HPD are concerned, together with the initialization of a SPD.

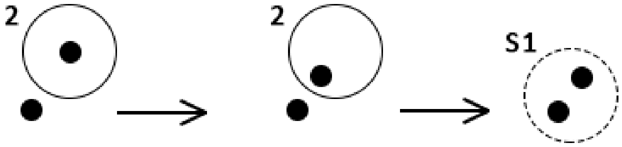


Figure 3.6 aCRD E02 Type 1 Conjunction Event Scheme.

This sub-event links one transient diffusion event (E02) to a reaction event (E10). This sub-event results in the break of one existing HPD and the buildup of a new SPD. To the defect cluster, this sub-event turns the cluster from HPD protected status to a released status, finished with the turn of all released clusters to the SPD protected status. In the style of algorithm, this event is described as:

1) Running Green's function solver for the neighboring protected walker and sending it to the new location inside the original HPD (Local Synchronization);

2) Breaking the respective hard PD;

3) Checking location, (*IF*) VERY close;

4) (*THEN*) building soft PD, go to reaction event (type 10 event).

In this event, the Green’s function solver is required, also with several location checks to initial the buildup of a new SPD.

Adding to this type of conjunction events, the type 2 conjunction event of transient diffusion (type 02 event series) is described as Fig. 3.7 for a special case in the event network, where the black dots represent defect clusters while the closed circles are HPDs. The circle noted as S1 with dash boundary represents a SPD overlapped with a HPD. In this sub-event, one defect cluster and its HPD are concerned, together with the updating of one SPD.

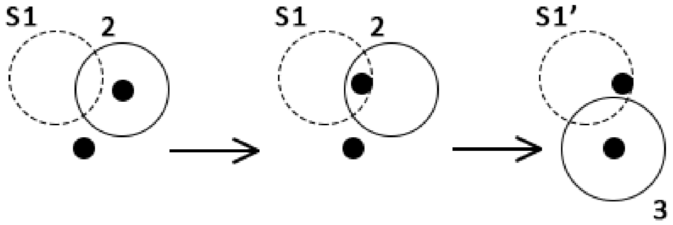


Figure 3.7 aCRD E02 Type 2 Conjunction Event Scheme.

This sub-event links one transient diffusion event (E02) to a transient diffusion event (E02), at the same time, triggers the update of SPD (E10 event series). This sub-event results in the break of one existing HPD and the update of a new SPD. To the defect cluster, this sub-event turns the cluster from HPD protected status to a released status, finished with the turn of several released clusters to the HPD protected status and several to the SPD protected status. In the style of algorithm, this event is described as:

1) Running Green’s function solver for the neighboring protected walker and sending it to the new location inside original HPD (Local Synchronization);

2) Breaking the respective HPD;

3) Checking location, (*IF*) inside of a SPD;

4) (*THEN*) go to reaction event (type 10 event), rebuilding the non-overlapping hard PD for another walker;

5) Rerunning the Green's function solver, sampling time of arrival for the hard protected walker, updating the event queue, processing to next event.

In this event, both the Green's function solver and the local reaction solver are required, also with several location checks to initial the update of SPD.

Three sub-events described in this section form the part of network initialized from a released cluster and forms the E02 transient diffusion event series. In all, this series serve as important conjunctions to different type of events in the events network of dynamics.

3.1.1.1.3. Reaction (E10) Series

In reaction event series, the core is the main event and its update logic, where conjunction events follow the similar rules as major diffusion (E01 series) and transient diffusion (E02 series).

An example of the main event of reaction (type 10 event series) is described as Fig. 3.8, where the black dots represent numbers of defect clusters, the circle with dash boundary represents a SPD and the closed circles are HPDs.

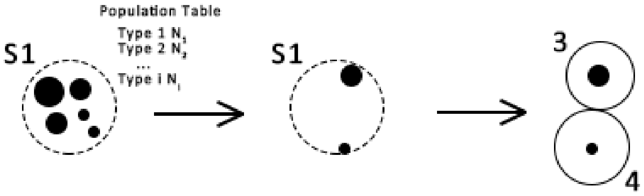


Figure 3.8 aCRD E10 Main Event Scheme.

This sub-event links one reaction event (E10) to random event in event queue. This sub-event results in the break of one existing SPD and the buildup of several HPDs. To the defect cluster, this sub-event turns the clusters registered in their SPD member list from SPD protected status to a released status, finished with the turn to the HPD protected status. In the style of algorithm, this event is described as:

1) Running the *N*-body possibility solver, sampling final status for the system within destination time, setting event stamp;

2) Reaching first event in global events queue;

3) Sending walkers at final states to respective stochastic locations and breaking the soft PD;

4) Rebuilding non-overlapping hard PDs for all transported walkers;

5) Rerunning the Green's function solver, sampling time of arrival for the hard protected walkers, updating the event queue, processing to next event.

As a different system comparing to diffusion, the algorithm above is not designed to be processed in consequence. It can be separated into several different stage inside the algorithm flowchart: initialization of event (first step in the algorithm), execution of event (2nd to 5th step in the algorithm). Between 1) and 2), update of SPD member list is able to be triggered from sub-events from other series (e.g. E02 type 2 conjunction event), this will only modify the information transported to local reaction solver without changing the event algorithm. Though Fig. 3.8 only shows a final status with 2 clusters, E10 event series are designed to be compatible to random number clusters in the final status of one SPD, which depends on the solver.

In aCRD algorithm, SPDs are not allowed to overlap each other though they are allowed to overlap with HPDs. Instead, SPDs will merge into a single SPD with a full list of their members once the overlap condition is occurred. This design avoids the forming of the network with large number of conjunction events from E10 main event with unknown final clusters. However, it is still possible for an E02 event to take place and to serve as the beginning of an event chain. In this way, the link between E10 to E02 can be regarded as one of the conjunction events in E10 event series.

3.1.1.1.4. Dissociation (E11) Series

Dissociation event (E11) series is a set of functional events, which helps to perform the dynamics better with optional subroutines. In general, this requires separate solvers and independent procedure. In this section, a basic design of dissociation event series is introduced. One important feature for dissociation event series in current aCRD design is that this series is only enabled in HPDs, as dissociation is able to be serving as part of reaction solver and no need to run independently in SPDs.

The main event of reaction (type 10 event series) is described as Fig. 3.9, where the black dots represent defect clusters while the closed circles are their HPDs.

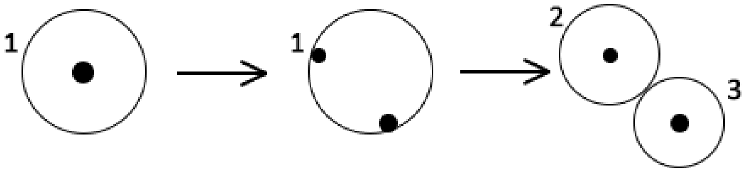


Figure 3.9 aCRD E11 Main Event Scheme.

This sub-event links one dissociation event (E1) to a random event in event queue. This sub-event results in the break of one existing HPD and the buildup of several new HPDs. To the defect cluster, this sub-event turns the cluster from HPD protected status to a released status, finished with the turn of all released clusters to the HPD protected status by other HPDs. In the style of algorithm, this event is described as:

1) Running a Boltzmann solver for all walkers in HPDs, sampling times of dissociation and registering in global events queue;

2) Running Green's function solver for the walker related to first event and sending it to the new location inside its original HPD (Event Synchronization);

3) Rerunning the Boltzmann solver to have final status and randomly releasing final walkers with certain distance away from each other, breaking the HPD;

4) Checking location, (Not) too close to another HPD or inside of a SPD;

5) Rebuilding non-overlapping HPDs for all walkers;

6) Rerunning the Green's function solver, sampling time of arrival for the newly protected walkers, updating the event queue, processing to next event.

In this event, both the Green's function solver and a Boltzmann solver to sample the final status to dissociation are required, also with several location checks to avoid the execution of conjunction events.

There are two types of conjunction events in type 11 event series. They are designed to link this event series to other event series.

The type 1 conjunction event of dissociation (type 11 event series) is described as Fig. 3.10, where the black dots represent defect clusters while the closed circles are their HPDs. In this sub-event, multiple defect clusters and their HPD are concerned.

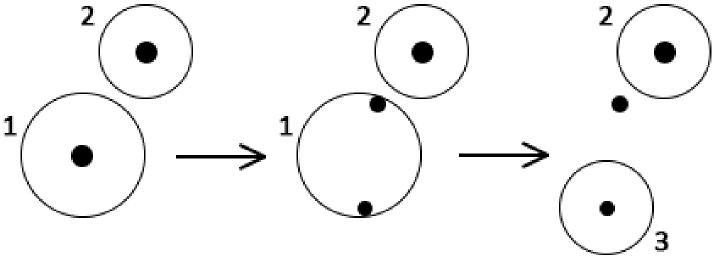


Figure 3.10 aCRD E11 Type 1 Conjunction Event Scheme.

This sub-event links one dissociation event (E11) to a transient event (E02). This sub-event results in the break of one existing HPD other than the original HPD which protected the dissociated defect cluster. To the defect cluster, this sub-event turns the clusters from HPD protected status to a released status, finished with the turn of part of released clusters to the HPD protected status and leave the rest at released status. In the style of algorithm, this event is described as:

1) Running a Boltzmann solver for all walkers in HPDs, sampling times of dissociation and registering in global events queue;

2) Processing Event Synchronization;

3) Rerunning the Boltzmann solver to have final status and randomly releasing final walkers with certain distance away from each other, breaking the HPD;

4) Checking location, (*IF*) too close to another HPD;

5) (*THEN*) go to transient diffusion (E02 event series), rebuilding the non-overlapping HPD for another walker;

6) Rerunning the Green's function solver, sampling time of arrival for the hard protected walker, updating the event queue, processing to next event.

In this event, both the Green's function solver and a Boltzmann solver to sample the final status to dissociation are required, also with several location checks to introduce the link to other event series.

Adding to this type of conjunction events, the type 2 conjunction event of reaction (type 11 event series) is described as Fig. 3.11 to link with SPD events, where the black dots represent defect clusters while the closed circles are HPDs. The circle noted as S1 with dash boundary represents a SPD overlapped the original HPD. In this sub-event, a SPD is concerned with the possibility to update its member list.

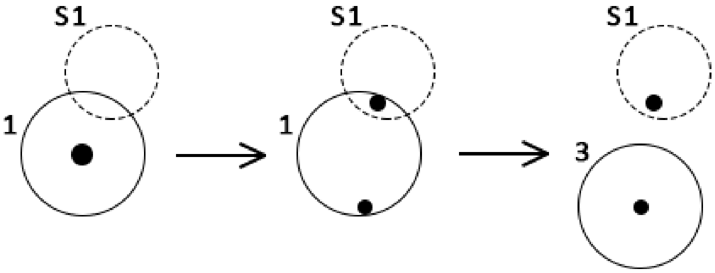


Figure 3.11 aCRD E11 Type 2 Conjunction Event Scheme.

This sub-event links one dissociation event (E11) to a reaction event (E10). This sub-event results in the update of existing SPD. To the defect cluster, this sub-event turns the clusters from HPD protected status to a released status, finished with the turn of part of released clusters to the HPD protected status and leave the rest at released status to be protected in SPD. In the style of algorithm, this event is described as:

1) Running a Boltzmann solver for all walkers in HPDs, sampling times of dissociation and registering in global events queue;

2) Processing Event Synchronization;

3) Rerunning the Boltzmann solver to have final status and randomly releasing final walkers with certain distance away from each other, breaking the HPD;

4) Checking location, (*IF*) inside of a SPD;

5) (*THEN*) go to reaction (E10 event series), rebuilding the non-overlapping HPD for another walker;

6) Rerunning the Green's function solver, sampling time of arrival for the hard protected walker, updating the event queue, processing to next event.

In this event, both the Green's function solver and a Boltzmann solver to sample the final status to dissociation are required, also with several location checks to introduce the link to event inside SPD.

Three sub-events described in this section form the part of network initialized from a dissociation inside HPD and forms the E11 dissociation event series. In all, this series serve as a functional event and a link to different type of events in the events network of dynamics.

3.1.1.1.5. Irradiation (E20) Series

Similar as E11 event series, irradiation event (E20) series is also a set of functional events, which helps to enlarge the function in aCRD algorithm to simulate the scenario related to the research of radiation damage in material. As an event series to introduce new clusters to the simulation space, it doesn’t require any new solvers but does require independent procedure.

This sub-event links one irradiation event (E20) to a random event in event queue. Based on the location of new defect inside the system in a released status, different types of conjunction events will be triggered. Generally, E20 event series can be regarded as a multi-entry event to most types of event series and follow their algorithm.

Based on the design, it is able to handle any type of irradiation profile, from Frenkel pair to damage cascade. However, as the damage cascade requires a large area of space to share with the same clock and process its initial evolution, it requires a hybrid-driven system to PROJECT aCRD which is not in consideration at current stage.

3.1.1.2. Flowchart in the Style of Events Network

In this section, a set of flowcharts combining with major diffusion (E01 event series), transient diffusion (E02 event series), reaction (E10 event series) and dissociation (E11 event series) are provided and separate the whole procedure into three different stages: initialization, event execution and global synchronization.

As this section is focus on the algorithm logic, the details in solvers would not be included in this section, related contents recording the details applied in prototype code systems are included in Section 4.

Fig. 3.12 is the flowchart to the initialization stage. This stage consists of the procedures from the beginning of aCRD algorithm to the initialization of global event queue.

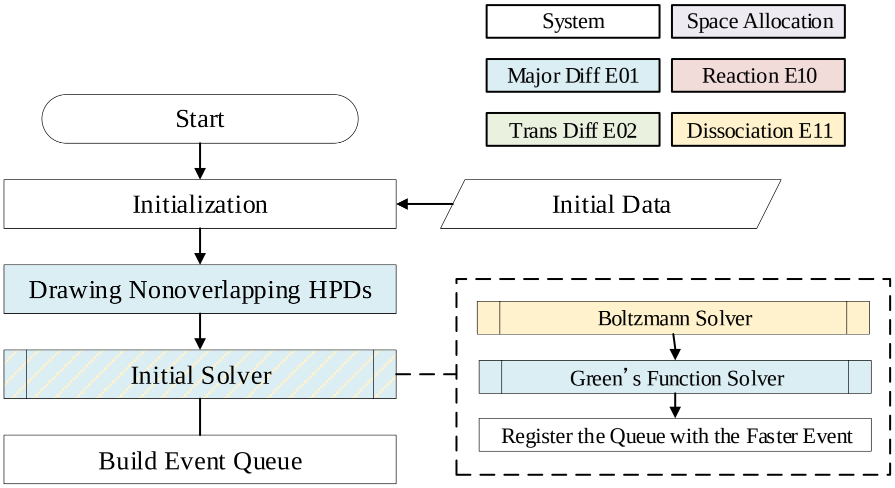


Figure 3.12 aCRD Algorithm Flowchart: Initialization.

In this stage, the system is initialized and all user defined inputs are introduced to aCRD system including the cluster profiles, parameter libraries and the profile for simulation box.

With the original defect profile, the first generation of HPDs are built. Based on the diffusivities of each defect cluster, original non-overlapping HPDs are introduced to cover the simulation space. In this step, the space allocation module is also required to assign the space to each HPD. In order to maximize the advantage of first-passage process, HPDs should be built as large as possible.

After the built of initial HPDs, the solvers are introduced to the system. Based on the definition of initial event types, different solvers might be introduced in this step, e.g. Green's function solver for E01 major diffusion event series and Boltzmann solver for E11 dissociation event series. The first event to be triggered inside each HPD is dependent on the sampled event time from each solver, only the first event is considered.

The last step in initialization stage is to build the global event queue. In this queue, all events in aCRD are registered in the order of event time stamps. Once the queue is ready, the preparing to execute the event-driven cluster diffusion and reaction dynamics is finished. From there, the system is proceed to next stage.

Fig. 3.13 is the flowchart to the event execution stage. This stage combines all different types of events into an event-driven algorithm.

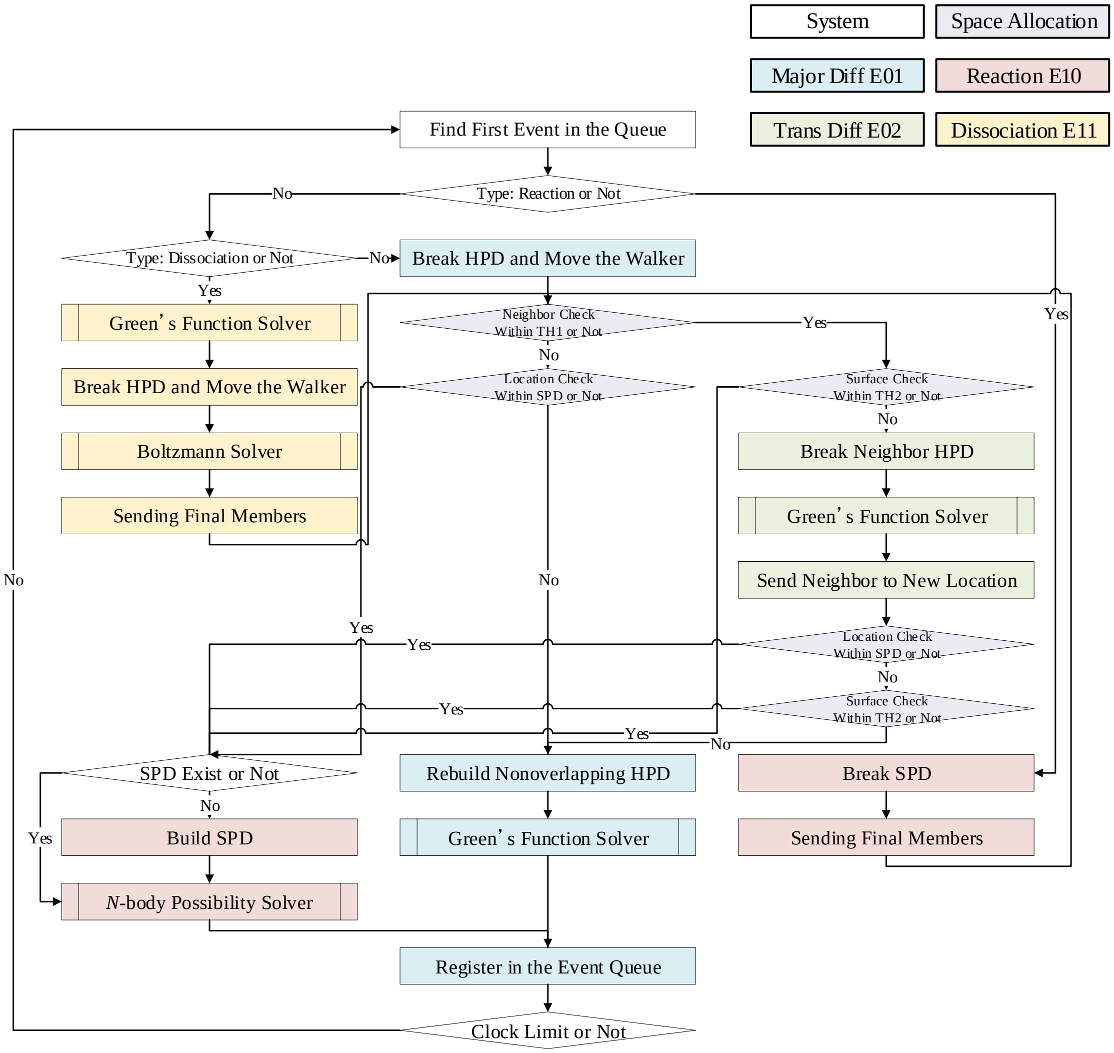

System
Space Allocation
Major Diff E01
Reaction E10
Trans Diff E02
Dissociation E11
Find First Event in the Queue
Type: Reaction or Not
No
Yes
Type: Dissociation or Not
No
Break HPD and Move the Walker
Yes
Green's Function Solver
Break HPD and Move the Walker
Boltzmann Solver
Sending Final Members
Neighbor Check Within TH1 or Not
Yes
No
Location Check Within SPD or Not
No
Surface Check Within TH2 or Not
No
Break Neighbor HPD
Green's Function Solver
Send Neighbor to New Location
Location Check Within SPD or Not
No
Surface Check Within TH2 or Not
Yes
Yes
Yes
Yes
No
No
SPD Exist or Not
No
Yes
Build SPD
N-body Possibility Solver
Rebuild Nonoverlapping HPD
Green's Function Solver
Break SPD
Sending Final Members
Register in the Event Queue
Clock Limit or Not
No

Figure 3.13 aCRD Algorithm Flowchart: Event Execution.

In this stage, events are executed and created controlled by the events network. For each time, the first event in the event queue is selected to be proceed. Once an event is processed to the execution stage, multiple logical selections lead the system to follow the algorithm described in events network. During this stage, defect clusters are running their dynamics following the rules of events network and the simulation box proceeds to next time stamp. In Fig. 3.13, the irradiation (E20) events series are not included.

After each event, all defect clusters are designed to be protected by a SPD or a HPD. Based on the time stamp to current event, the world clock is checked to judge whether proceeding next event or moving the system to next stage.

If an event reaches the wall time based on user setting, the global synchronization stage will be proceed. Fig. 3.14 is the flowchart to the global synchronization stage. This stage consists the procedures from a global synchronization to the finishing of aCRD system.

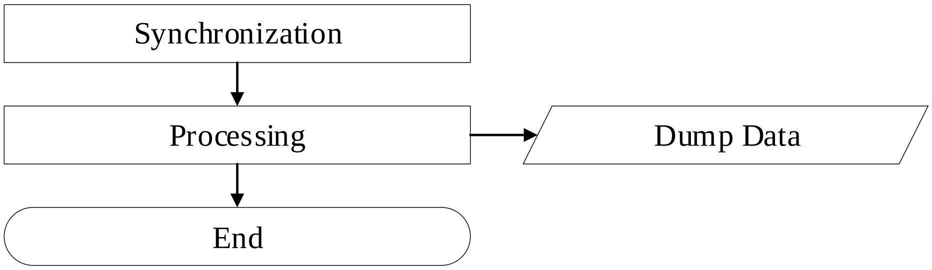


Figure 3.14 aCRD Algorithm Flowchart: Global Synchronization.

In this stage, all defect clusters still exist in the simulation box are proceed to a global synchronization. As the clock check is triggered by the last event, only the defects related to that event are updated to the current world clock, in the other word, all other defects are still have time stamps earlier than world clock. However, as their events have not be triggered, an individual procedure is required to handle the synchronization event. For the defects protected in HPD, transient diffusion events (E02) are required to transport these defects to somewhere within their HPDs. For the defects protected in SPD, reaction events (E10) are required to sample both final status and final locations within their SPDs. In this way, all defects are synchronized to the world clock and there is no protected defects any more.

After the global synchronization, the system is proceed to next step which is the dump of useful data. As an extra function, the simulation box is able to be synchronize with certain intervals, this is the global synchronization function. Once this function is required to synchronize the whole system, a multi-pass calculation will take place after this step. The system will shift back to the initialization stage and proceed to a new world clock once the global synchronization is done.

The flowchart above shows the logic of process an event-driven dynamics. However, as shown in Fig. 3.13, once the dynamics become complicated, the event flow will be quite intricate and hard to do algorithm maintenance or function upgrades. And it will cause troubles for the development of code systems once several single events are triggered continuously in one single event.

In order to handle this challenge, as one of the most important upgrades during the development of aCRD algorithm, the area synchronization multi-stage (ASMS) procedure is developed and introduced to aCRD algorithm to modify the event execution logic.

3.1.2. Area Synchronization Multi-Stage Procedure

Area synchronization multi-stage (ASMS) procedure is developed to modify the event execution logic and to avoid the troubles due to multiple event triggered in one event flow. Generally, ASMS consists of two main stages, which is releasing stage and rebuilding stage. Fig. 15 describes the general concept of ASMS procedure.

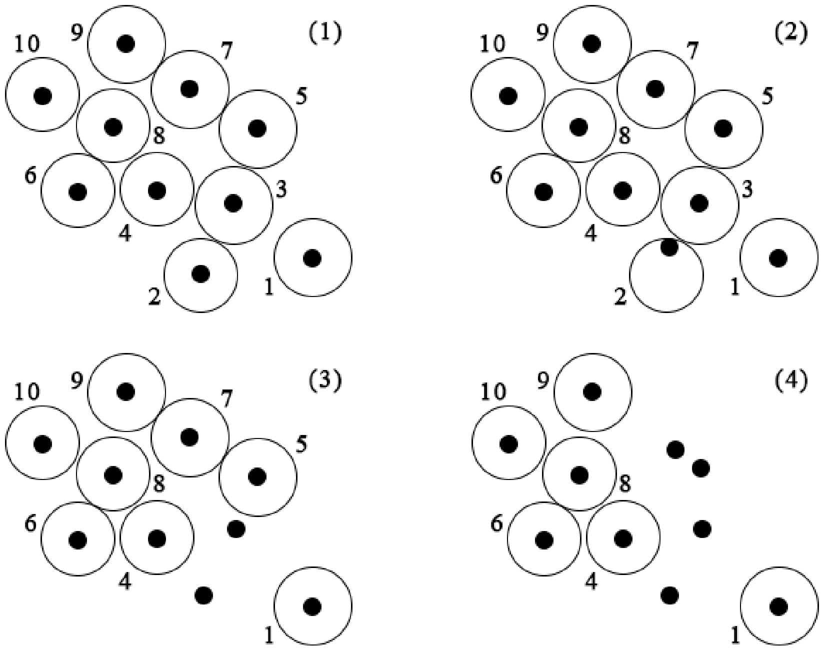


Figure 3.15 Area Synchronization Multi-Stage Stage-1 Scheme.

From Fig. 3.15 (1) to Fig. 3.15 (2), it shows a general event beginning from a major diffusion event in HPD-2, as the walker protected in this domain is diffused to the close boundary near HPD-3, a transient diffusion is triggered. In all, this is the same event knowns as type 1 conjunction event in major diffusion (E01 event series). However, from Fig. 3.15 (1) to Fig. 3.15 (2), it shows the case once the walker in HPD-3 is diffused to the near surface of HPD-5, which is quite possible in a simulation condition with high density.

In this way, another transient diffusion will be trigger, which may form another trigger for different events. In this way, the system shows a requirement to handle unknown events in one event flow which is not possible for the original events network described in the last section. This results to the initialization of the design of ASMS procedure.

In ASMS's first stage, the releasing stage, all released walkers are considered with backtracking to their neighbors. The system will keep checking all released walkers whether any HPD is nearby or not. The backtracking will not be terminated until all released walkers are proved to be away from any existing HPD, as shown from Fig. 3.15 (3) to Fig. 3.15 (4).

Once all released walkers are checked and verified to be away from HPDs, the system will proceed to rebuilding stage. The boundary checking will be applied between the end of releasing stage and rebuilding stage. If any of these released walkers satisfied with specific boundary condition, they will be handled separately and not proceed to the rebuilding stage.

In the rebuilding stage, all released walkers will be protected by new domains again, they may be linked to either SPD protected status or HPD protected status by several sub-stage checking rounds.

The second stage scheme is described in Fig. 3.16. At the beginning of rebuilding stage, all walkers released in the first stage are sent to be check with all effective SPDs. In this step, all released walkers are checked with their locations. If any of these walkers are inside of effective SPDs, these walkers would be send to the SPD update sub-event (E10 event series). This links ASMS to the same procedure as event network procedure described in Section 3.1.1.1.

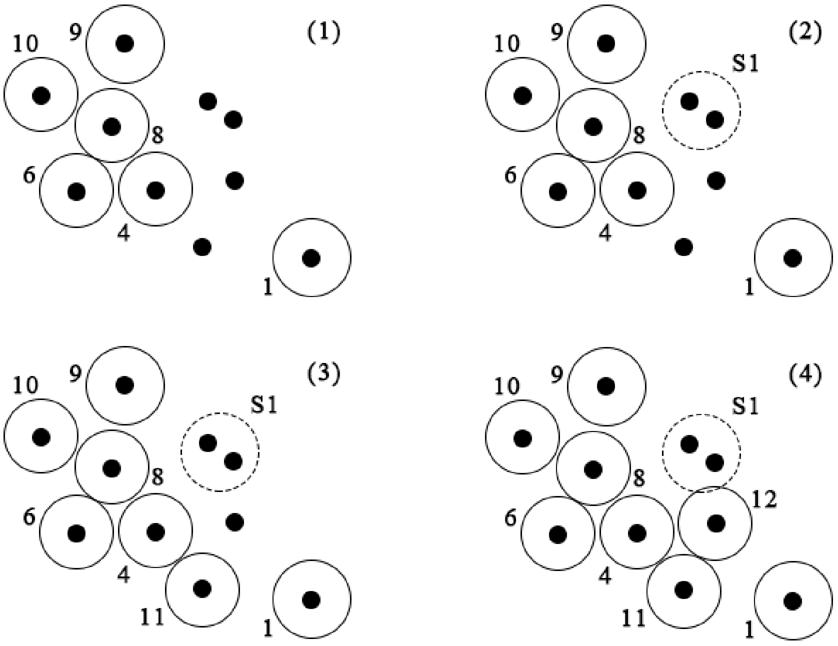


Figure 3.16 Area Synchronization Multi-Stage Stage-2 Scheme.

After the SPD checking, the surface distance between all walkers still haven't been protected by domains are examined. If any of these walkers are very close to one another, a new SPD will be built to protect both walkers, as shown in Fig. 3.16 (2) where a new SPD noted as S1 is built to protect two walkers released during first stage and failed to be protected in any SPD during the SPD checking. This step is similar to the beginning parts of general E10 reaction event as described in Section 3.1.1.1.

The final step in ASMS stage-2 is the protection with HPDs to all walkers still haven't been protected. In this step, the shortest distance from the walker to the surfaces of any existing HPDs, the shortest reduced radius to another released walker based on each other’s diffusivity and the shortest distance to the centers of any existing SPD are calculated and compared to each other. New HPD to this walker is built based on the largest value in these distance. The calculation of three different distance and the protection form one checking round. In this step, for each checking round, only one walker will be protected by its new HPD, as shown in Fig. 3.16 (3) and Fig. 3.16 (4) to avoid logic error from the distance calculation.

For any walker in stage-2, once it is protected by either SPD or HPD, the related solver is introduced to create a new event and inserting this event to the event queue.

After all walkers become protected again, ASMS reached its end and next event will be initialized.

3.1.2.1. Events Network with ASMS Procedure

With the introducing of ASMS, the logic of events network is greatly modified. All sub-events now share with the same entrance (event execution) to ASMS and have the same exit (new event creation) from ASMS to next event. Fig. 3.17 shows the scheme between different types of events and ASMS procedure.

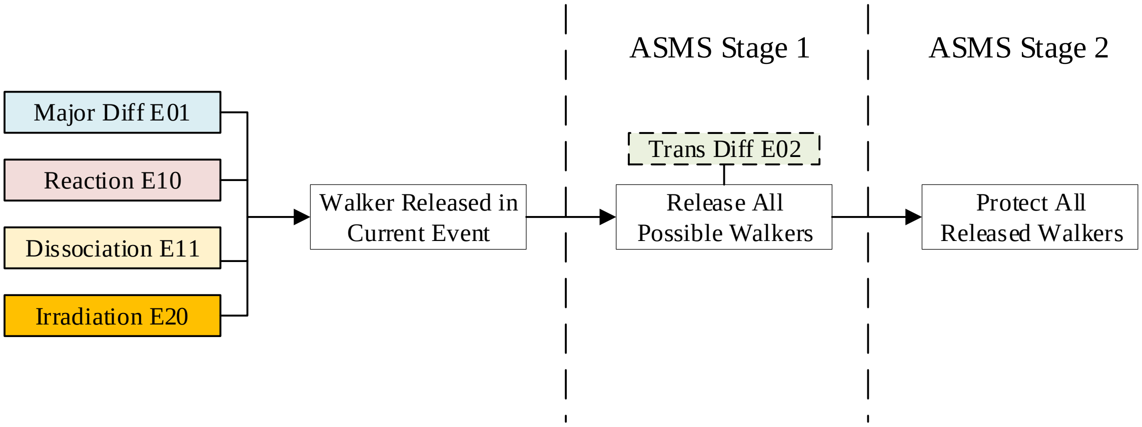


Figure 3.17 Scheme of ASMS and Different Event Series.

With ASMS procedure, both event released walkers and new introduced walkers are able to be treated equally as “released walkers”, this feature guarantees the flexibility to upgrade aCRD algorithm with open boundary and changing population system.

At the same time, different event series no longer occupy a large space in main logic, the system modules become more independent from the events modules. However, once basic dynamics is modified, ASMS procedure may require updating to fit with new dynamics.

Sharing with the same entry to different events also means the system will be easy to upgrade to new functions for different scenario.

3.1.2.2. Flowchart in the Style of ASMS

As described above, the introducing of ASMS greatly modifies the logic of event execution stage in aCRD algorithm. The flowchart of this stage will be modified to the one shown in Fig. 3.18 from the complicate one shown in Fig. 3.13.

In the flowchart shown as Fig. 3.18, the initialization stage and global synchronization stage follow the ones described in Fig. 3.12 and Fig. 3.14. It is able to tell that with ASMS procedure, the complicate logical judgement and space allocation are no longer required.

Major diffusion event (E01) series, reaction event (E10) series, dissociation event (E11) series and irradiation event (E20) series all enter ASMS first stage with different conditions but same entry. Meanwhile, ASMS 2nd stage offers a general exit with new event creation and event queue registration. Both features make the algorithm friendly to program and friendly to read.

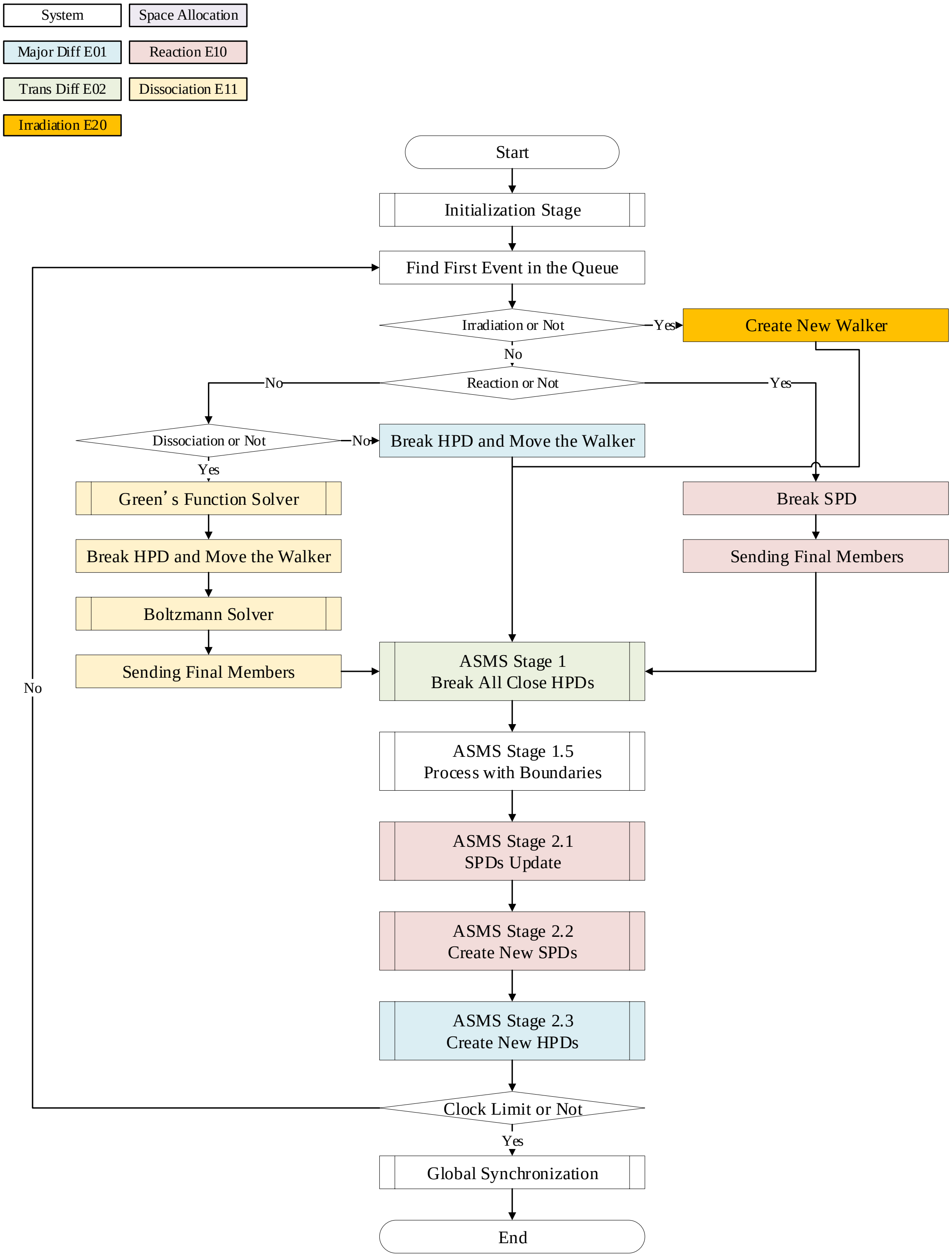


Figure 3.18 aCRD Algorithm Flowchart with ASMS Procedure.

3.2. Code Systems

Based on the algorithm describing the defect clusters diffusion and reaction dynamics, the code systems are designed to form the framework to apply this algorithm and to test algorithm’s capability. In order to meet with the requirement on flexibility, a multi-level structure is designed.

3.2.1. Design and Framework Structure

In the design of aCRD code systems structure, a three-level architecture is introduced. As shown in Fig. 3.19, the code system consists:

- Logical control level (LCL)
- Logical execution level (LEL)
- Event level (EVL)

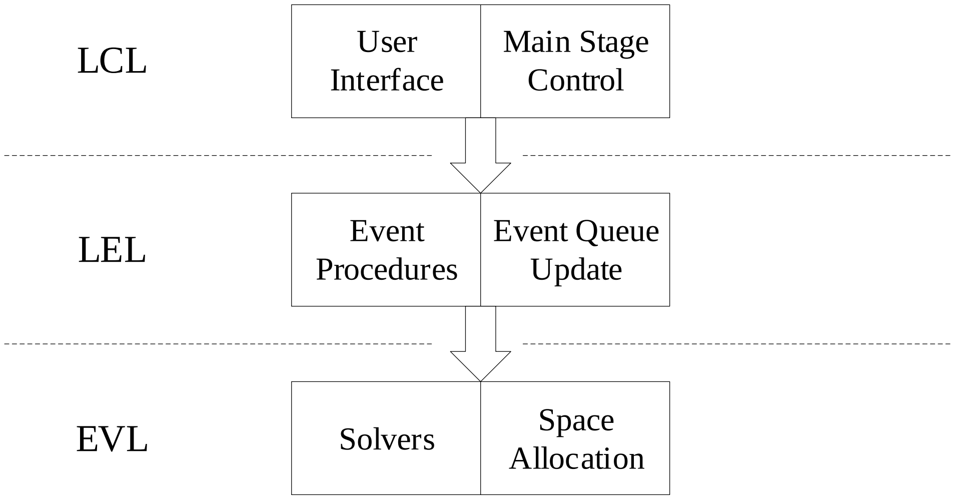


Figure 3.19 Scheme of aCRD Three-Level Structure.

The highest level, LCL, consists of a user interface and the code for main stage control. It is designed to handle the progress management. Once certain flags handled in LCL related to the procedures in each stage are triggered, the code systems move forward to next stage, e.g. a finishing-reading flag indicating the success during user data reading procedure to start the initialization of HPDs, a finishing-initialization flag indicating the success during initialization procedure and the built of global event queue to begin event execution stage. The selection of next event and the clock check procedure are also regarded to be in this level.

The second level, LEL, consists of the procedure for each event series and the update of event queue. Any event selected in LCL is designed to be sent to the code in this level and perform the related procedures. Most code for ASMS procedure are recorded in this level. Generally, the code in this level manager the modification of walkers' (defect clusters') information and the execution of events network. At the end of each event procedure, the code to create new events, to update global event queue, to sort the event queue and to update the world clock are also recorded in this level.

The code in LCL and LEL are maintained in the main procedure and its related tool procedures in aCRD code systems. In both levels, only the interfaces to different solvers are recorded, the solving of Green's function or any sampling method is not designed to be in this level.

The last level, EVL, consists of the solvers and the code to allocate space in simulation box. Communicating with the main process through the interface in LEL, the information to modify walkers' information are calculated in EVL and sent back to LEL. The code for this level are designed to be maintained in independent procedures from the main procedure.

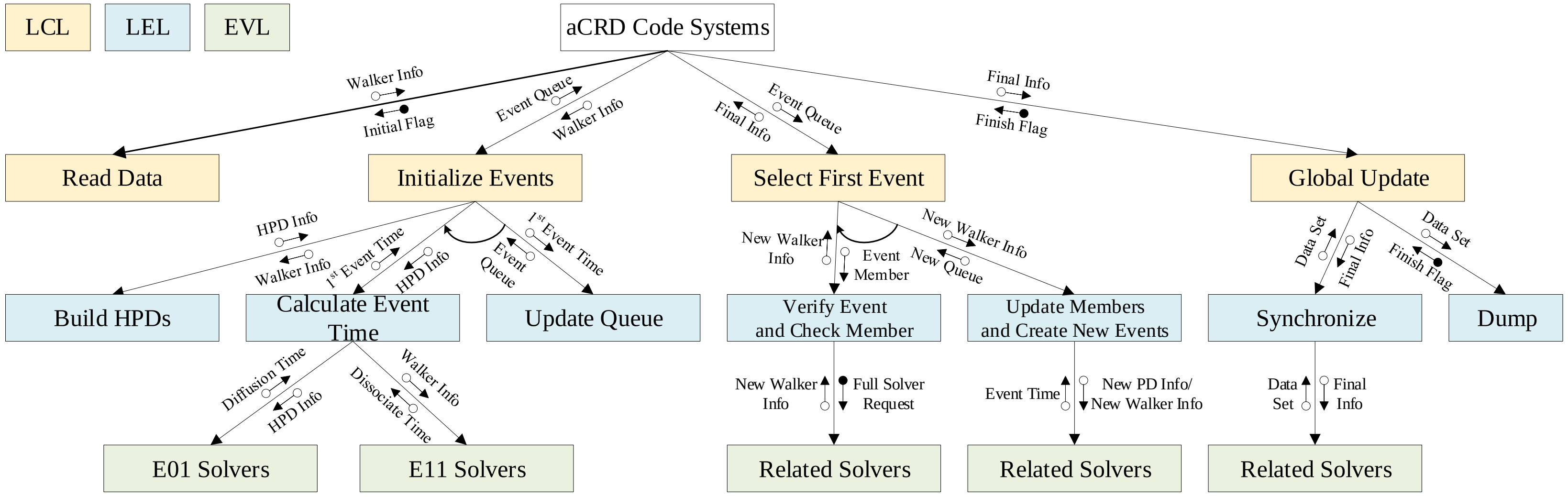


Figure 3.20 aCRD Program Structure Diagram.

This design separates the code systems into several levels, which separates the functions to individual parts and makes the system easy to maintain.

A program structure diagram to all three levels of aCRD code systems is provided in Fig. 3.20. The first two branches belong to the initialization stage in aCRD algorithm, the third branch belongs to the event execution stage while the last branch belongs to the global synchronization stage.

In the initialization part, Fig. 3.20 shows the separation of EVL and LEL. With independent levels, the aCRD structure is flexible to apply different user-defined initialization events for different applications.

In this diagram, the initial event only consists of major diffusion (type 01 event series) and dissociation (type 11 event series). For a system with much more complicated initial profile, reaction (type 10 event series), and even other event types which haven't been defined in current aCRD are able to be added to EVL to be considered during initialization. This type of structure upgrade only requires a modification on LEL logic and an upgrade on EVL solvers.

As a matter of fact, designed to be a compatible version to multi-pass calculation which requires the simulation box to be synchronized with certain time intervals, the prototype code systems included in the current version aCRD already enables the buildup of SPD (enable type E10 event series) during initialization.

In the event execution and global synchronization part, Fig. 3.20 shows how the solvers are separated from higher levels of code systems. With LEL code handling the information update to the walkers (defect clusters) in the simulation box, the creation of new events which relies on solvers is not relied on any calculation with the code in LEL. This feature guarantees a handy interface for user-defined event. Once the space allocation module is built for aCRD system, all procedures to create new events would be independent from the main procedure, however, this is not included in current version of aCRD and will not be discussed in this section.

For the multi-pass calculation function in aCRD, the last three branches shown in Fig. 3.20 are designed to be repeated with certain time intervals. Once multi-pass calculation function is enabled, all walkers in the simulation box will be synchronized after certain time intervals to talk with each other and to rebuild their domains. This function will abandon part of the speed advantage on asynchronous event-driven system with better compatibility to the system with changing members.

Based on the features on other asynchronous KMC systems, the statistic behavior of asynchronous KMC algorithm will match with general synchronous KMC after a large number of events. However, this hasn't been benchmarked with aCRD or any system with large member changes due to severe irradiation conditions. In this way, the multi-pass calculation function is embedded to current aCRD to offer more options to future researchers.

More features and details are needed to explain the logic inside the branch of event execution stage about how the program is designed to begin the execution, to communicate with solver modules and space allocation modules, to create new events and to communicate with global queue, a data flow diagram is provided in next section for this branch.

3.2.2. Event Subroutine Data Flow

In this section, a data flow diagram for event subroutine is provided in Fig. 3.21 and discussed to explain the features in aCRD system.

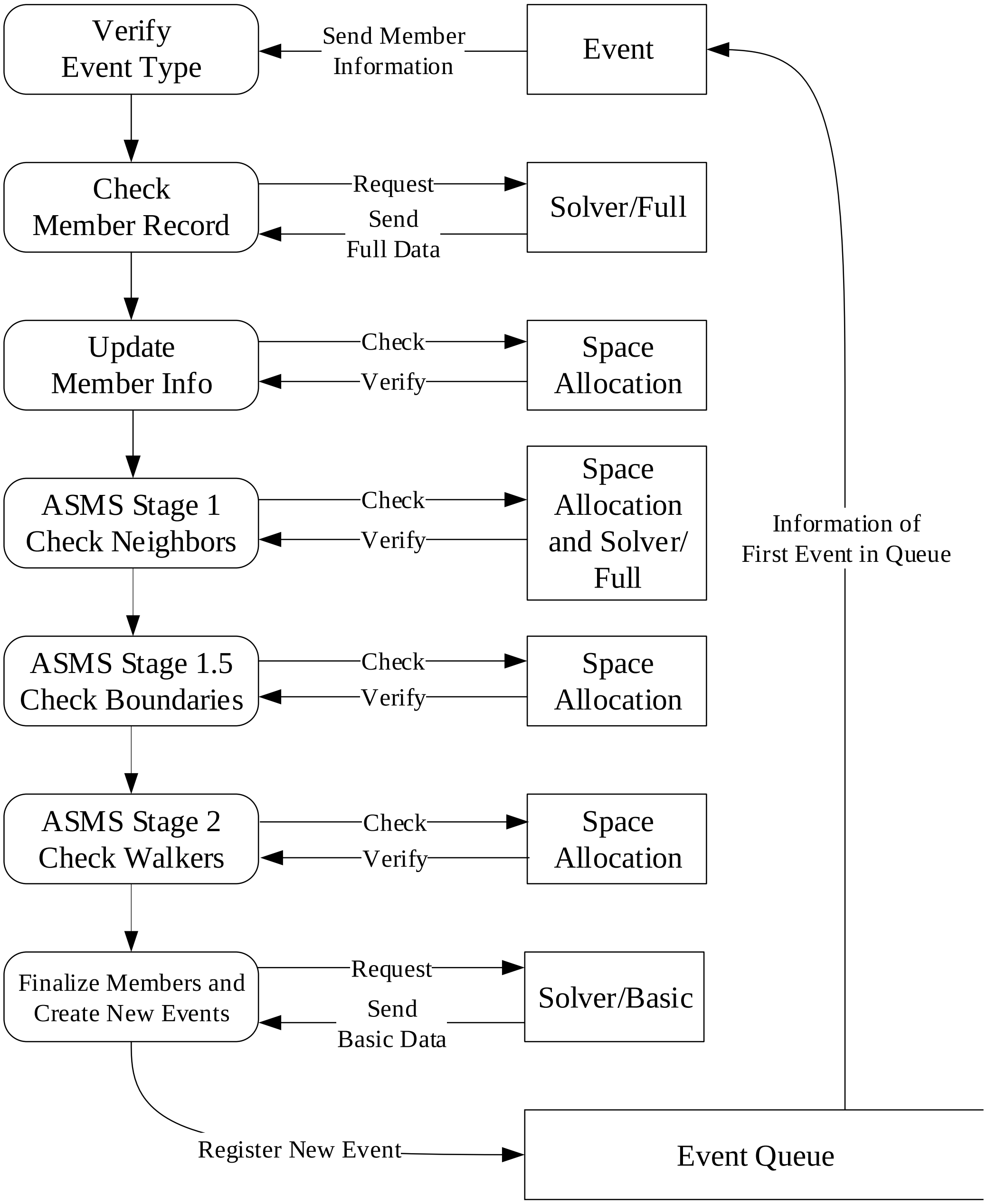


Figure 3.21 aCRD Event Data Flow Diagram.

After an event is selected to be executed, the related information about event type, the walkers and domains will be sent for processing. After identified the event type, the related processing methods and related solvers are processed. In this step, the solvers are requested to provide the final status for walkers related in this event. With the information from solvers, the walkers and domains are updated with these information, this step also relates to the space allocation modules. After this step, ASMS is initialized and processed with its multi-stage procedure. During these steps, related solver modules and space allocation modules are requested by main procedure for each triggered event. Once ASMS procedure reaches the end, new events are registered in event queue. This step requires the time information provided by solver modules, however, a full solution is not required as the location and final status information are not required.

The data flow diagram described how the space allocation module is independent from the main data flow. This design ensures the space allocator is able to process special requirement without affecting the main data flow, e.g. adding a function to simulate multi-dimension diffusion with same data flow. This feature also enables the capability to define special structures with the same data flow.

Meanwhile, Fig. 3.21 also demonstrates how solvers modules are independent from the main data flow. As the main data flow requests data from solver modules through general interface, it is quite handy for developers to upgrade solver modules independently and to enable the capabilities of aCRD code systems on different applications.

Another important feature demonstrated in Fig. 3.21 is the separation of solver interfaces for different steps in main data flow. In the processing of single event and the processing in ASMS, the full version of solvers are requested to provide the information of final status to related walkers, while the basic version of solvers are requested to provide only event time information during creating new events. This feature enables the system to be on-hold for solvers to get their solution during the processing of other events. This design directly improves the calculation efficiency, and also enables a paralleling solving upgrade for future version, which is ideal for the development of an implementation on supercomputing architecture.

4. PROTOTYPE CODE SYSTEMS

A Prototype code system is built with C++ in this project as a performance demonstration of PROJECT aCRD and a verification to the design and concept described in Section 3.

In this section, the details on the prototype code system are introduced. The frame of code systems, the assumptions applied in current prototype main procedure and solver modules and details on events dynamics for different version of prototypes are included in this section.

Sets of data with different scenario are analyzed in Section 4.2 and Section 4.3. They perform as the verification to the basic design and the evidence of meeting with current goal to the project.

4.1. Prototypes for Verification

At current stage, two prototype code systems have been developed for the functional verification.

The first prototype is the version to perform a 1-D logical and functional verification. In this prototype system, the event dimension for all defect clusters is set to be 1-dimension only, HPDs and type I SPDs are introduced to the calculation. An absorption boundary is applied to remove any defect cluster reaching the edge of simulation box to simulate the effect of grain boundaries. As a logical verification code system, this prototype system only concerns with a dynamics consists of 1-D diffusion and annihilation, which means two walkers will both be removed from the simulation once they collide with each other. In this version, system modules, partial library module, major diffusion (type 01 event series), transient diffusion (type 02 event series) and partial reaction (type 10 event series) event modules and a prototype Green’s function solver module are maintained in a close loop with several functional tests. It is designed to obtain certain data to prove that the system is free from major bugs. At the same time, this version is supposed to be applied to check the behavior of event-driven system on its efficiency to simulate a diffusion system.

The second prototype is the version to perform a 3-D full functional verification. In this system, the event dimension for all defect clusters is set to be 3-dimension, full functioned HPDs and type I SPDs are introduced to the calculation. The absorption boundary is applied to remove any defect cluster reaching the edge of simulation box to simulate the effect of grain boundaries. As a functional verification code system, this prototype system concerns with a full dynamics of 3-D diffusion and reaction, consists of diffusion, coalescence and a partial dissociation. At the same time, irradiation is also enabled in this version. It means two defect clusters will diffuse in a 3-D style, coalescent with each other if they reach a certain surface distance and dissociate with certain possibility depending on their size. Meanwhile, the system is introducing new defects with certain time intervals. In this version, system modules, partial library module, major diffusion (type 01 event series), transient diffusion (type 02 event series), reaction (type 10 event series), dissociation (type 11 event series) and irradiation (type 20 event series) modules together with a prototype Green’s function solver module and a prototype local reaction solver are maintained in a close loop with functional tests. This version is designed to obtain certain data to show the mesoscale capability to simulate defect clusters' evolution, also to test the multiscale capability for the code to simulate large scale evolution.

In this section, details in these prototype code systems are introduced. Limited by the size, the source code will not be attached to this dissertation. Instead, the logic flow of code systems will be described in the coming sections.

4.1.1. Member Variables in Prototypes

In prototype, the variables applied to describe walkers (defect clusters) and domains are recorded in Table 4.1 and Table 4.2 respectively.

Table 4.1 Table of Walker Variables Applied in aCRD Prototype Code Systems.

Variables	Content	Variables	Content
ID	A unique number assigned to each walker.	Energy	Reserved to describe the temperature distribution, in keV.
Type	Describe cluster type.	Diffusivity	Cluster diffusivities, in nm^2/s.
Component	Describe cluster size.	Status	Describe the clusters' protection status.
Center Location	Describe the center location of clusters, in nm.	Region ID	Log the related region's ID to next event.
Radius	Describe clusters with sphere assumption, in nm.	Timestamp	Describe the last event to this cluster, in s.

In Table 4.1, cluster type is assumed to be a value as 0 for vacancy clusters, 1 for interstitial clusters or -1 for clusters which have been disabled from system. Cluster status is assumed to be a value as 0 for released clusters, 1 for clusters protected in SPDs or 2 for clusters protected in HPDs.

Table 4.2 Table of Domain Variables Applied in aCRD Prototype Code Systems.

HPD Variables	Content	HPD Variables	Content
ID	A unique number assigned to each HPD.	Event Type	Describe the type of next event.
Walker ID	Log the related walker's ID to next event.	Event Time	Log the next event time, related to the initial time of HPD.
Walker Type	Log the related walker's type to next event.	Event Card	Log the related event card registered in event queue.
Radius	Describe HPD with sphere of line assumption, in nm.	Final Location	Log the diffusion location for E01/E02 event series.
Center Location	Describe the center location of HPD, in nm.	Final Status	Log the final status for E11 event series.
SPD Variables	**Content**	**SPD Variables**	**Content**
ID	A unique number assigned to each SPD.	Center Distance	The distance between two walkers at SPD's two centers.
Current Member	Log the number of current members in this SPD.	Event Type	Describe the type of next event.
Final Member	Log the number of final members in this SPD.	Event Time	Log the next event time, related to the initial time of SPD.
Member List	Log all the walkers exist in this SPD.	Event Card	Log the related event card registered in event queue.
Center Locations	The center location information for the 2 centers of SPD.	Final Status	Log the status of final members for E10 event series.

In Table 4.2, event type for HPD is assumed to be 1 for major diffusion event to break HPD, 11 for dissociation event to break HPD or -1 for HPDs which have been disabled. For major diffusion or transient, the variable final location is reserved to storage the calculation data returned from related solver modules. For dissociation event, the variable final status is reserved to storage the calculation data returned from E11 solver. In current version prototype, dissociation event series are not designed to be effected by independent events, instead, they are combined with major diffusion event series. Details are introduced in Section 4.1.3.3.

For SPDs, event type is assumed to be 10 for reaction event to break the domain or -1 for SPDs which have been disabled. The variable final status is reserved to storage the calculation returned from E10 solvers. In current version, type I SPDs are designed with two centers very close to each other based on the location of two center defect clusters forming this SPD. With certain radius from both centers, the corresponding SPD is built and maintained. Information about the centers are recorded in the variables center locations and center distance.

For domains, variable event card is designed to link domains to event cards in global event queue.

In current version of prototype, the memory to storage all defect clusters and domains information are designed to be preassigned in source code with certain maximum volume. Memory allocation and releasing are not included in current version in order to have a better tracking of code systems' behavior in details through debugging. This feature makes current prototype hard to reach a simulation with large number of clusters.

4.1.2. Main Stage System

1D prototype code system and 3D prototype code system share with the same main stage system. It follows the aCRD algorithm with ASMS and the program structure diagram described in Section 3.

The main stage for prototypes are built in a main function which consists of initialization stage, event execution stage and global synchronization stage.

In initialization stage, the program is proceed with the following steps: system initialization, reading user data, space initialization, building global events, creating irradiation event and sorting global event queue.

System initialization handles the loading of dynamic-link libraries containing solver modules and library modules. At the same time, integrity check is proceed. A function list of current version aCRD prototype is attached in Appendix B as reference.

After the loading of all functions, program is proceeded to read user data and system configuration files. During this step, original profile of initial defect clusters, dimension information and the limitation of world clock are loaded to the program from external files.

Next step is to initialize the simulation box and to assign domains to each original cluster. This step follows the flowchart shown in Fig. 4.1. The first check round in this step is to check the overlap of clusters, if any cluster overlapped with another, one of the clusters will be disabled. After this check round is to check whether any cluster is very close to another. Surface distance of two clusters within a certain threshold (defined by users) will initiate the building of a new SPD. The last part in this step is to build HPDs to protect any cluster which is still active but hasn't been protected by any domain. The initial size of HPDs for clusters depend on their diffusivities, a cluster with larger diffusivity will be assigned with a larger HPD.

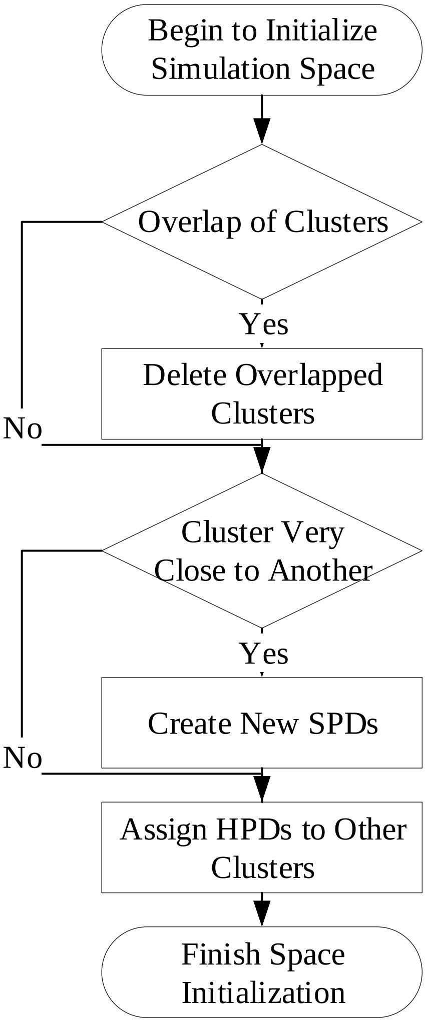

Begin to Initialize Simulation Space
Overlap of Clusters
Yes
No
Delete Overlapped Clusters
Cluster Very Close to Another
Yes
No
Create New SPDs
Assign HPDs to Other Clusters
Finish Space Initialization

Figure 4.1 aCRD Prototype Space Initialization.

After the initialization of simulation space, the program moves to the step to build global events, to be more specific, the step to create the event card for each domain. In this step, the functions from solvers are applied to the program for the first time. For each domain created in last step, related solvers are applied to calculate the first-passage event time. These information will be recorded to each domain. Event cards are created once events finished with the calculations of event time. For event cards, world clock time are recorded instead of the event time recorded in each domain. Event cards are linking to the event domains one by one.

Extra events are created based on the definition of irradiation condition by user. In current version, the prototypes only accept new clusters as multiple single defects landing at random location in the simulation box out of HPDs. For irradiation event (E20), irradiation interval and irradiation component number for each event are required to be defined, parameters as shown in Fig. 4.2. This step only creates one event card with defined time interval as the initial timestamp, new event card will be created once the last one is executed.

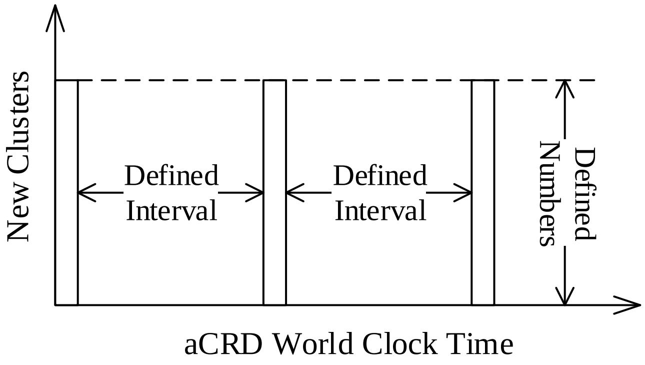


Figure 4.2 aCRD Prototype Irradiation Event Profile.

The last step in this stage is to sort the global event queue to the ascending order of event time (world clock time). This step initiates the global event queue. An event pointer is also initiated in this step which points to the first event in the queue.

After the sorting of global event queue, the program moves to next stage, the event execution stage. In this stage, the cluster diffusion reaction dynamics will be performed to evolve the simulation space.

In event execution stage, different dynamics are applied to evolve the simulation box depending on dimension information from input data. Detailed steps are recorded in Section 4.1.4. ASMS procedure is contained in this stage.

The last step in this stage is the same as the one in last stage, to sort the global event queue to the ascending order of event time (world clock time). This step will arrange all event cards create in ASMS procedure. The event pointer is designed to be moved to next event after the previous event has been processed.

Data dumping procedures are designed to be processed after the sorting of event list, all defect clusters which have not been disabled would be dumped to external file with information requested in different output files for different purposes.

Once the last event executed reaches the limit of world clock, the third stage is initiated. In this stage, global synchronization is applied to all defect clusters in the simulation box. In this step, all defect clusters protected in HPDs are forced to trigger E02 transient diffusion events and synchronize with the world clock. By sending the walkers to corresponding locations in their HPDs, these walkers is able to be regarded as released from any domain. For SPDs, the current version prototype will not modify its members' information, which means E10 reaction event series is not designed to be triggered during this step.

After the global synchronization, the system process a check for multi-pass calculation. If multi-pass is required, the whole system would go through all three stages once again. If multi-pass is not required, the program proceeds to another dumping procedure to storage the walkers' final information to external file. After this step, aCRD program releases all modules and exits.

4.1.3. Solver Subroutines

In this section, the subroutines for each solvers will be introduced with details and related approximations. In current aCRD, the solvers applied in major diffusion (E01 event series) and transient diffusion (E02 event series) related to solve Green's function are introduced together, the solver for reaction (E10 event series) and dissociation (E11 event series) are introduced with independent sections. Irradiation (E20 event series) does not rely on any solvers in current prototype.

4.1.3.1. Green's Function Solver Subroutines

As listed in Appendix B, subroutines related to Green's function solver consists of two functions for major diffusion event series and one function for transient diffusion event series. In this prototype, the sampling method follows the descriptions in Section 2.1.2 to set $C(t) \rightarrow S_1(t) = 2\pi^2 e^{-\pi^2 t}$ and several approximations described in this section.

One of the subroutines for major diffusion is the function to sample the exit time for a defect to exit its HPD. A general flowchart to this function is provided as Fig. 4.3. In this function, only the diffusivity information for the defect cluster and the geometric information of the domain is required to adjust the time unit to fit the general solving procedures in the solver. Once these information are sent to the solver, the program normalizes the unit of diffusion equation and processes a general Green's function random sampling method to obtain exit time (in the unit of L^2/D). After this step, the program turns the unit of time back to second and returns the value to main program.

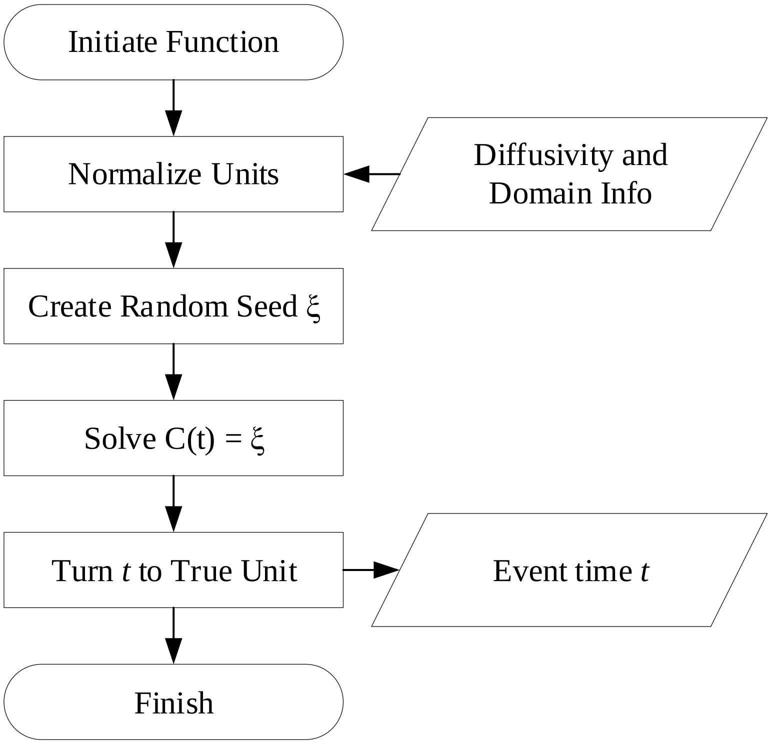


Figure 4.3 aCRD Prototype Main Diffusion Event Time Solver Flowchart.

With the information returned for the time sampling function, the main diffusion event card for this HPD is built and the program is able to proceed next commend.

Another function related to major diffusion event is the function to sample the exit location for the defect protected in its HPD. For this function, a uniform distribution assumption is referred in current solver. Refer to a simulation box without extra force field, chances for a set of diffusion events to transport the defect to different locations on the spherical HPD surface are regarded to be same. In this way, the function to sample the final location for an E01 major diffusion event randomly select a location on the surface by two random seeds sampling inclination angle and azimuth angle.

The function to sample the transportation location due to synchronization within HPD for E02 transient diffusion event series, in other word, sampling from $C(r;t)$, is not fully enabled in current solver, instead, a uniformly distribution assumption for the probability density inside the HPD is provided to create a demo solver performing in this prototype. In this solver, a totally random sampling is proceed to use three random seeds to sample radius, inclination angle and azimuth angle. Related work to provide a better solver is at top priority in current schedule of aCRD development.

4.1.3.2. Local Reaction Solver Subroutines

Local reaction solver is designed to be a solver applying independent logic to handle local multi-body interaction problem in SPDs. In original design, rate theory system, phase field system and hybrid-driven system (for type 2 SPD) are considered to be candidates of solution.

In current prototype, type 1 SPDs are designed to be break after interval of time, in this way, the solver designed to calculate the break time is no longer in need. Only the solver to sample final status is required in prototype.

Following a simple logic as shown in Fig. 4.4. This solver requires the members (defect clusters contained in this SPD) information from the original SPD, judges the main defect type taking control of the final defect resulted from interaction and returns the final member (defect clusters formed at the time to break this SPD) list to main program. The proceeding of interaction are determined by the calculation of final status, where different algorithms are applied.

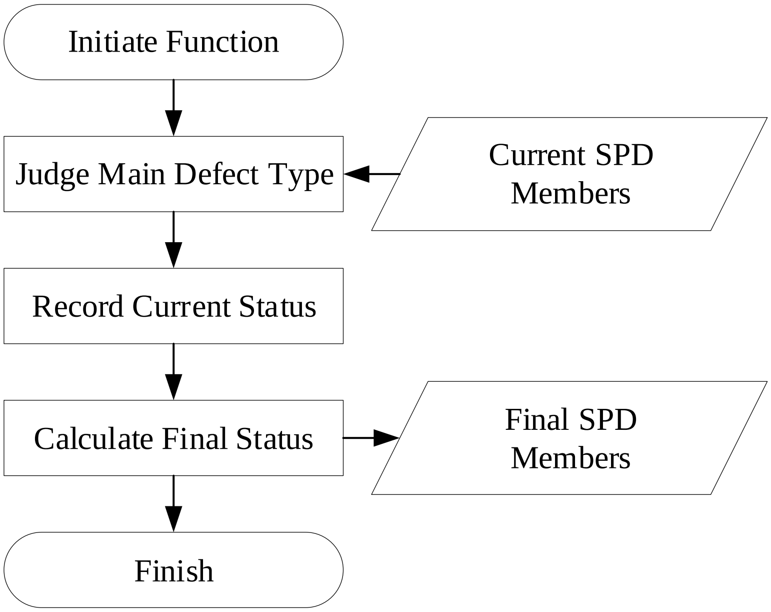


Figure 4.4 aCRD Prototype Reaction Solver General Flowchart.

In the demo version of current prototype code systems, this procedure is simplified to a simple adding of all existing members. With a very close threshold (0.5 nm) to build SPDs, the system considers all members in SPD will merge into one single defect cluster once SPD breaks. As reaction solver has a high impact on the building up of larger cluster, several plans are proceeding to further improve the solver to reach higher accuracy for local events. However, most data obtained for functional verification are based on current solver to merge all defect clusters in related SPDs.

4.1.4. Dynamics Events

In this section, details in event execution stage to perform the dynamics events are described in aCRD prototype latest version 1.0.4 (refer to the change log in Appendix A).

The event logic in this section based on the dynamics event network for 3-dimension system with diffusion, coalescence, irradiation and partial dissociation. The event pointed by the event pointer is selected to be processed for each round. During each round, the program processes the flow following Fig. 3.21. The logic flowchart is shown in Fig. 4.5.

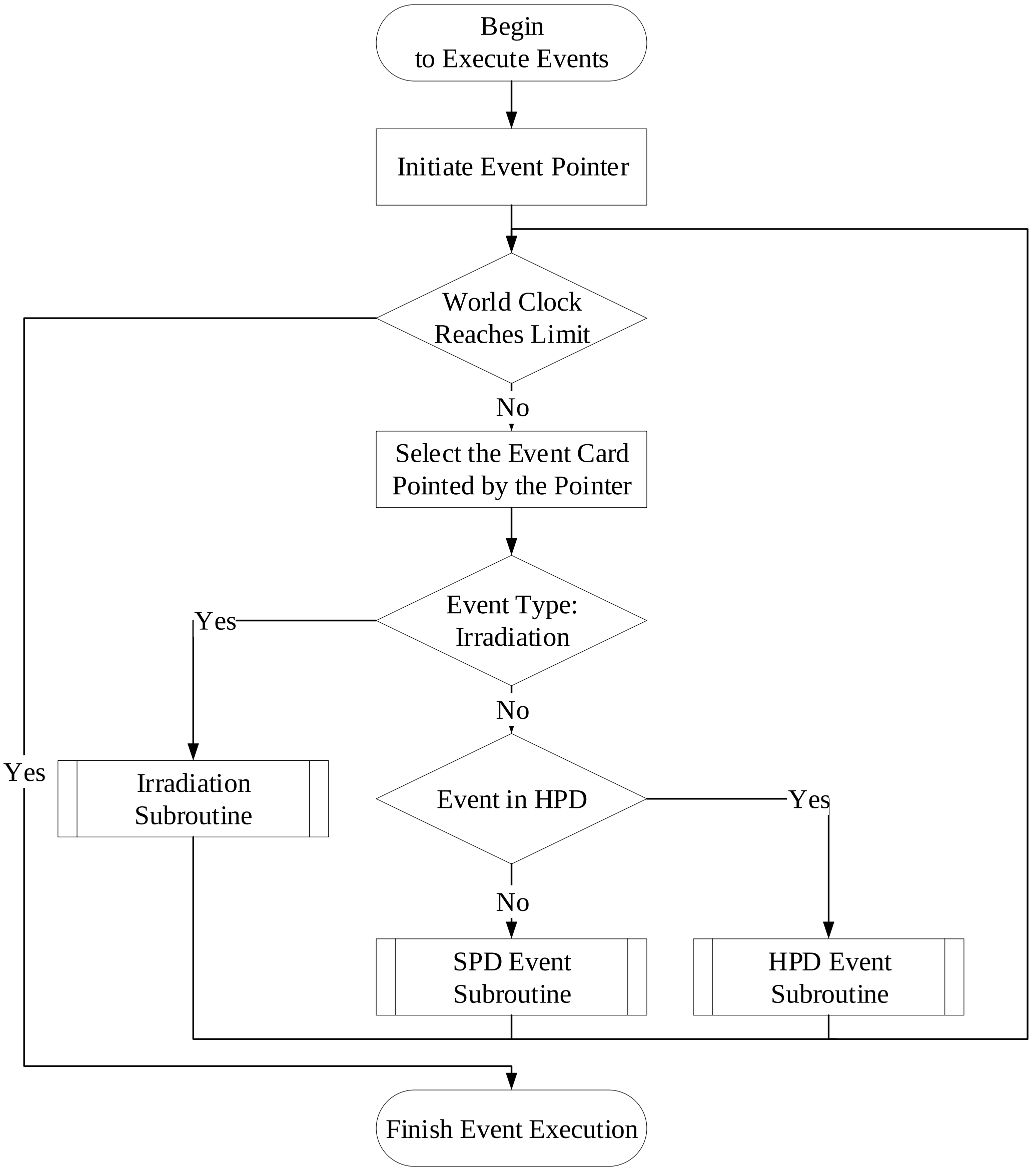


Figure 4.5 aCRD Prototype Event Execution Flowchart.

The irradiation event (E20 event series) subroutine follows the logic shown in Fig. 4.6. Multiple single defects based on user definition are introduced to simulation box to random locations out of any enabled HPD and away from any existed cluster.

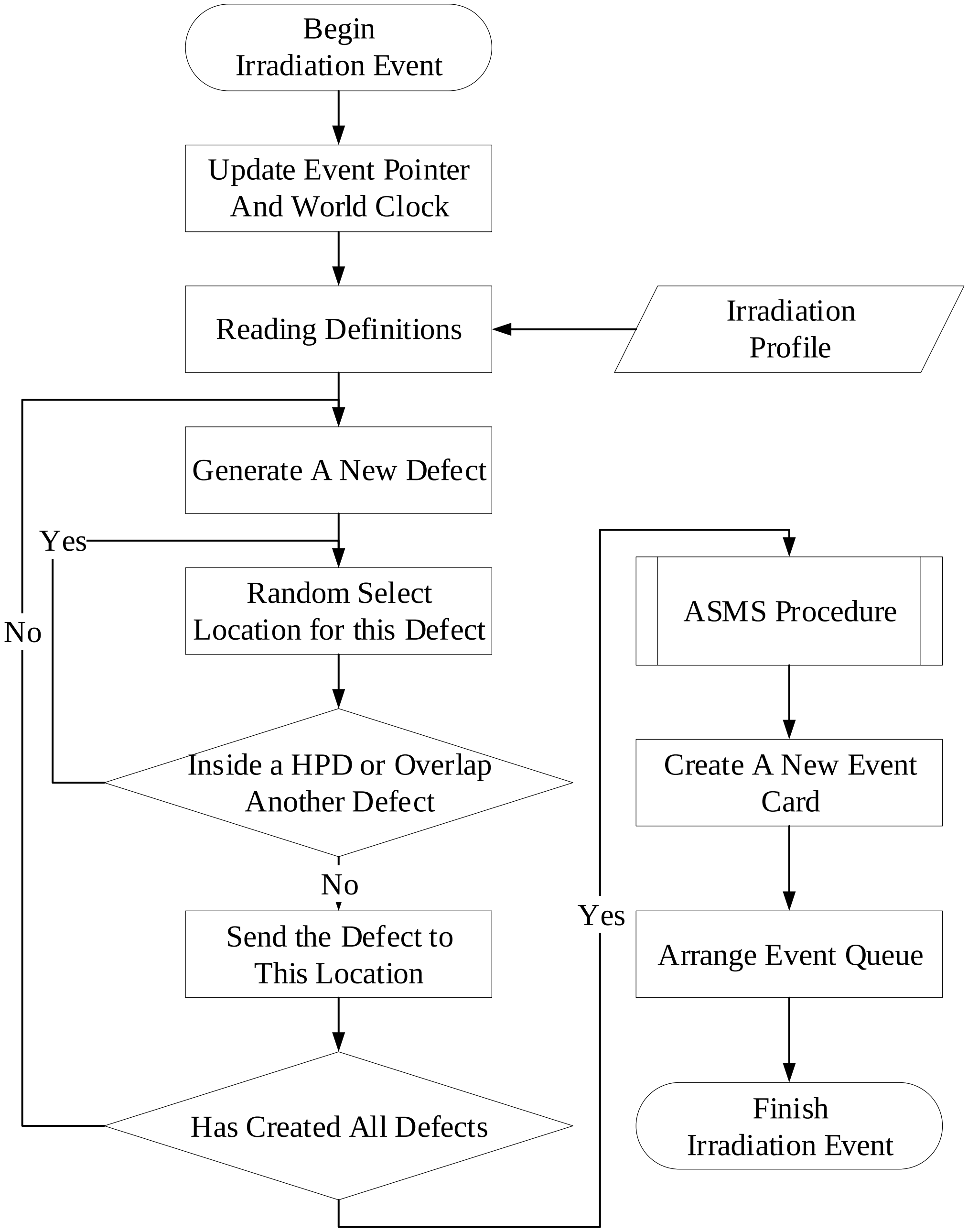


Figure 4.6 aCRD Prototype Irradiation Event Flowchart.

The HPD event (E01 and E11 event series) subroutine follows the logic shown in Fig. 4.7. In this subroutine, both major diffusion event and partial dissociation event are processed.

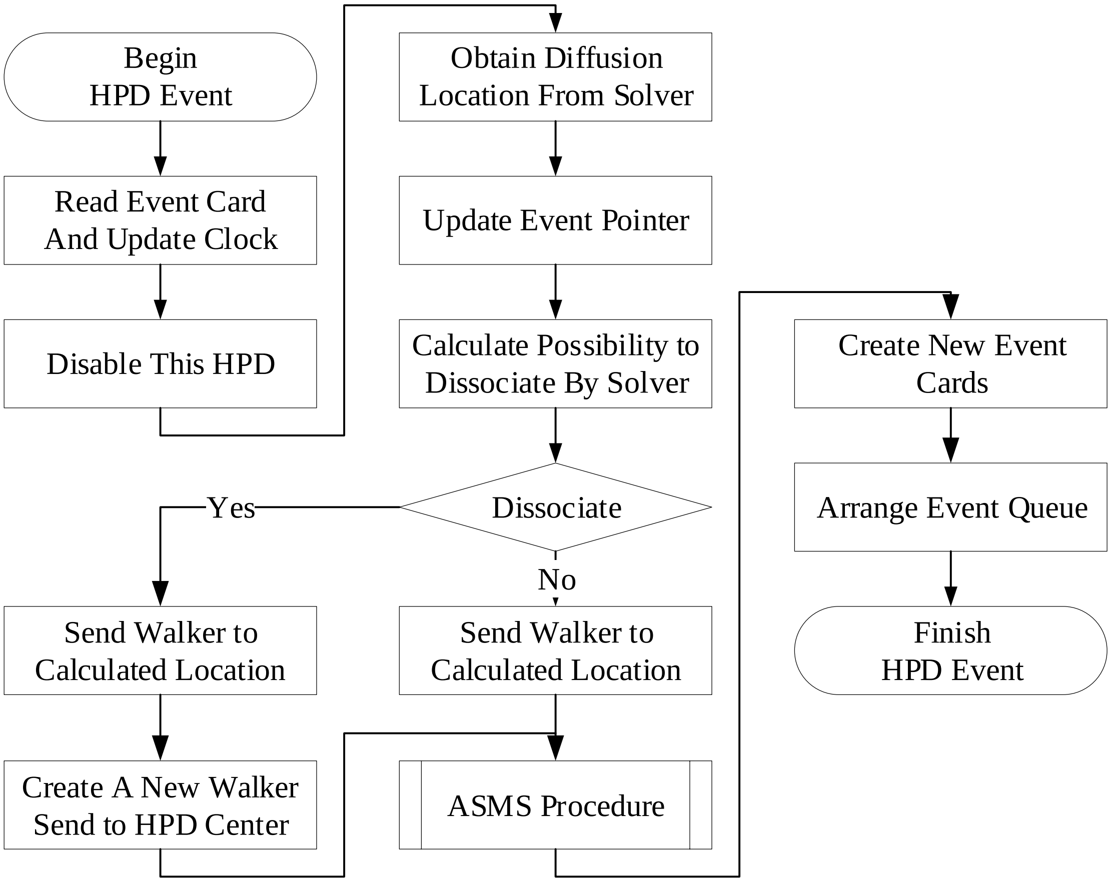


Figure 4.7 aCRD Prototype HPD Event Flowchart.

As shown in Fig. 4.7, the program reads current event card and domain information at the beginning of this subroutine and calls major diffusion solver module to calculate the location information for diffusion. After this step, the subroutine goes through a check through dissociation solver to judge whether the cluster protected in current HPD dissociate or not based on the size of cluster. If dissociation is triggered, the cluster with size N ($N > 1$) protected in this HPD would be transported to the calculated location on the boundary of domain with a changing on cluster size to N-1. A single defect would be created at the original location, which is at the center of this HPD. Both the cluster and the single defect are set to the released condition and registered in ASMS released cluster list pending for ASMS procedure. If dissociation is not triggered, the original cluster would be transported to the calculated location without a modification on cluster size.

The SPD event (E10 event series) subroutine follows the logic shown in Fig. 4.8. In this subroutine, reaction event is processed.

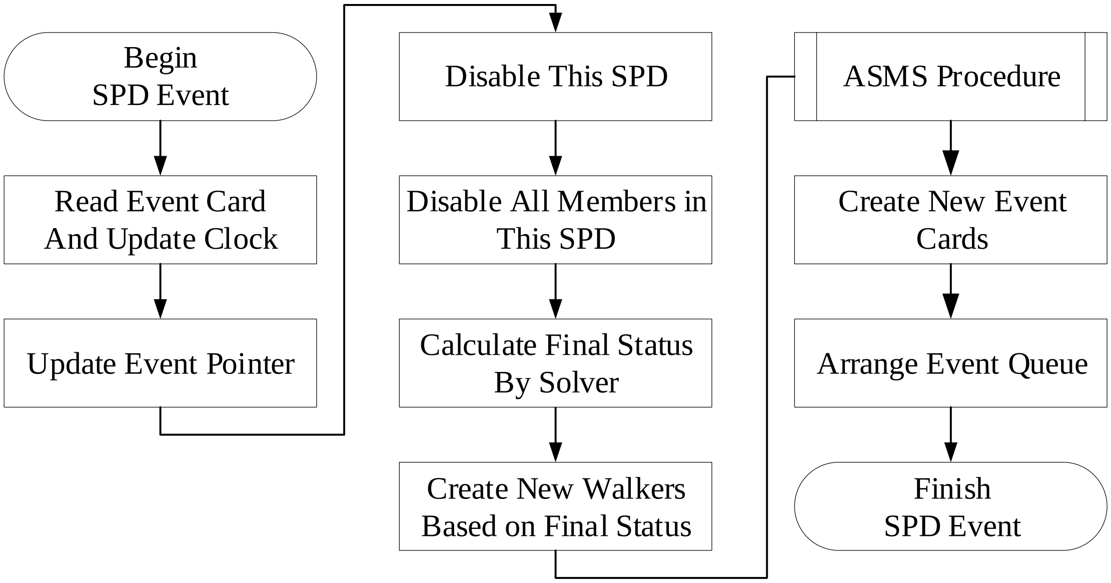

Begin SPD Event
Read Event Card And Update Clock
Update Event Pointer
Disable This SPD
Disable All Members in This SPD
Calculate Final Status By Solver
Create New Walkers Based on Final Status
ASMS Procedure
Create New Event Cards
Arrange Event Queue
Finish SPD Event

Figure 4.8 aCRD Prototype SPD Event Flowchart.

As shown in Fig. 4.8 and similar to the procedures in HPD event, the program reads the event card and domain information at the beginning of this subroutine. All current clusters in this SPD are disabled after the reading of event information. Then the reaction solver is requested to provide the information of final status of SPD based on the approximation described in last section. In this next step, the clusters from the calculation of solver are created with released status and proceed to ASMS procedure.

In these subroutines, several clusters are created or updated to the released status. ASMS procedure is proceeded at the end of each subroutine to modify all clusters and create new domains with new events adding to the global event queue. After ASMS procedure, the global event queue will be arranged again.

4.2. Data from 1D Prototype

In this section, data calculated in 1D system are collected and analyzed to provide a logical verification and to perform the code system behavior testing.

In 1D system, the dynamics with annihilation only is applied, which could be described as:

$$(A) + (A) \rightarrow 0$$

In other word, the walkers are designed to diffuse in a 1-dimension motion on a line, once two walkers collide with each other, both walkers are removed from the system. The data provided by this prototype are designed to verify the algorithm logic, to examine the collision possibilities change with the evolution of time and walker density, and to test the calculation efficiency at different walker density conditions.

A set of data testing are taken place on a 1D space with 50,000 nm length, applied with absorption boundary. Numerous walkers (single defects) are randomly assigned. Following with the dynamics of annihilation, these particles will: 1) diffuse out; 2) interact with another particle to annihilation (both removed from system). Fig. 4.9 shows the data from these calculation.

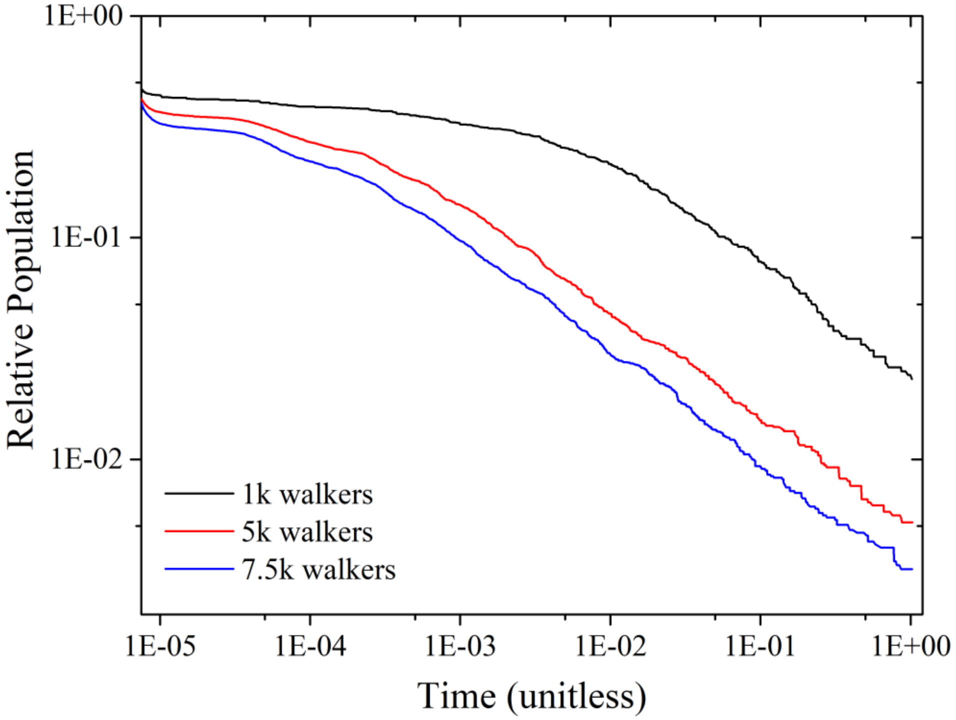


Figure 4.9 aCRD 1D Prototype Annihilation on 50000 nm with Closed Boundaries.

Fig. 4.9 shows the change of events frequency and the trend on walker removing in aCRD system. In Fig. 4. 9, the x-axis stands for the world clock in aCRD system, which is also known as the timestamp for each event. Here, the unit of time is arranged to relative time to the wall clock of aCRD for these calculation, which is 500s. The y-axis is the particle population relative to the original number. Both axes are arranged with log scale. Different walker numbers (different densities) are assigned to obtain these curves, where the black curve is obtained from 1,000 original walkers on 50,000 nm space, the red one is obtained from 5,000 original walkers on 50,000 nm space and the blue one is from 7,500 original walkers on the same space.

It is able to tell several features on current prototype from Fig. 4.9. The prototype successfully simulate the trend of population changes for annihilation and diffusion system. It shows a multi-stage dropping with scales change both with population and time, and the different behaviors at different scales. At the same time, Fig. 4.9 also demonstrates the difference of dropping with different densities. With a higher particle density, a system is more likely to diffuse its members out of boundary or interact with each other. However, limited by current functions, an interaction only testing is not able to be processed with a constant density which requires the periodic boundary.

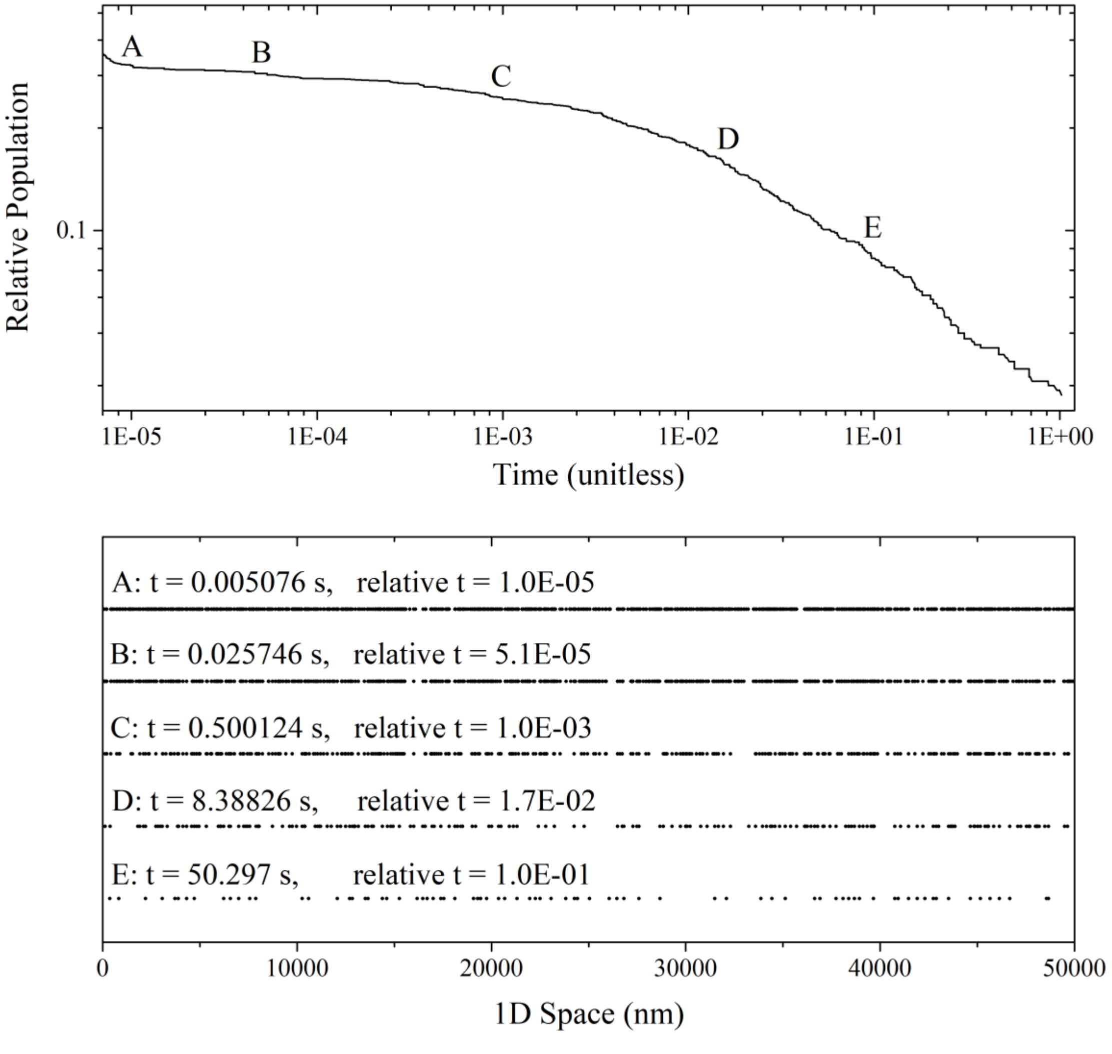


Figure 4.10 aCRD 1D Annihilation 1k Walkers on 50um at Different Time.

Fig. 4.10 further shows more details on the data in Fig. 4.9. Fig. 4.10 demonstrates the population dropping for the case of 1,000 walkers on 50,000 nm 1D space with absorption boundaries. The distributions of these walkers are noted on the second half of Fig. 4.10. The little dots in this figure represent the locations of each walker at different time. As noted in the first half of Fig. 4.10, five time stamps are selected to build the comparison between distributions, which take places at A: 0.005076 s (relative time 1.0E-5), B: 0.025746 s (relative time 5.1E-5), C: 0.500124 s (relative time 1.0E-3), D: 8.38826 s (relative time 1.7E-2) and E:50.297 s (relative time 1.0E-1). These set of data shows how the density in simulation box drops from its original and walkers evenly annihilated in 1D space. At the beginning stage, the system is maintained in high density, which initiates the rapid dropping of the member population with a high event frequency (lower time scale). As system density decreased to lower scales (larger space scales for events), the event rates also changes to a lower frequency (higher time scale).

Fig. 4.9 and Fig. 4.10 shows the evident to verify the design logic of aCRD. In 1D prototype, aCRD system shows the change of event frequency at different space scales and time scales. As the event number in aCRD doesn't increase at higher time scale with lower density, the calculation is able to be kept with a high speed to process the events and to advance the world clock. This shows the advantage on calculation efficiency at lower density conditions.

Calculation time is also logged to provide more data to analyze the behavior of aCRD prototype code systems. Fig. 4.11 shows the calculation time for different original densities. Logged time includes events processing and data dumping after each event.

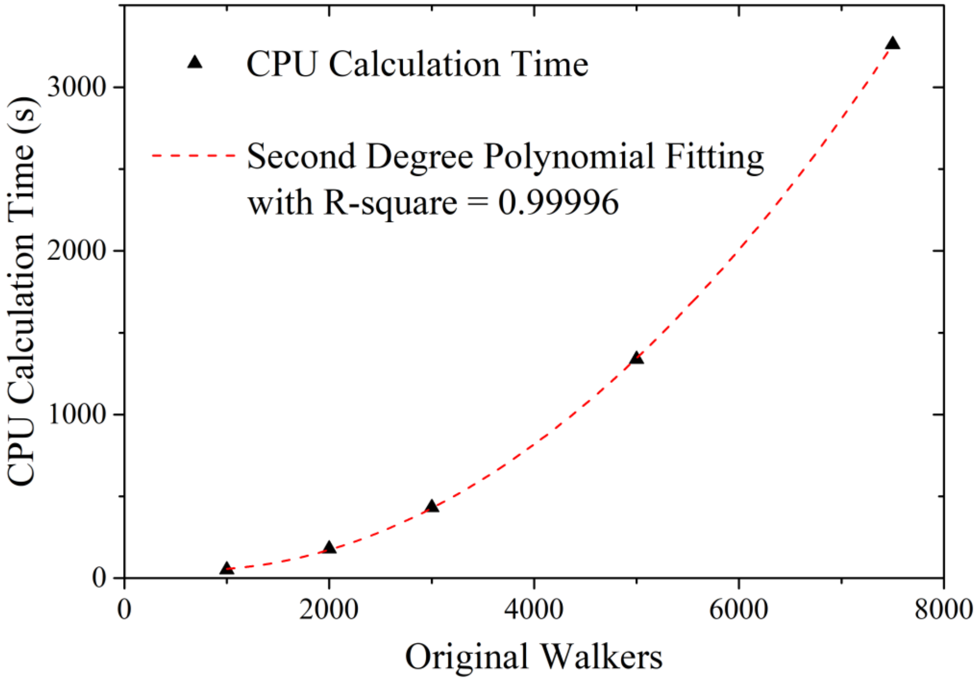


Figure 4.11 aCRD 1D Annihilation CPU Time with Different Densities.

Fig. 4.11 shows the real world CPU time taken to finish the calculation with different original walker numbers on 50,000 nm 1D space to advance to 500 s world clock in the aCRD prototype system. It shows the system behaves with O (N^2) which is the time complexity for current breadth-first [90] [91] [92] [93] domain searching parts. This indicates with further improvement on current system with advanced searching method to lower down the time complexity, the system is able to enable the calculation for larger system with walkers at a higher scale on its population. At the same time, the calculation time data in Fig. 4.11 also contains the time to dump data to external file after each event, which contains the code handling hard disk output with high time complexity and adds the hard disk writing time to the calculation time. As I/O speed of hard disk is far slower than CPU speed, the time data here does not reflect the real calculation speed for aCRD prototype.

Another issue arises from the 1D data is on the initialization part of current code systems. Refer to Fig. 4.9 and Fig. 4.10, the relative population drops from 1.0 (original walkers) to different numbers in very short time scale and enters the constant scale changing period (point A in Fig. 4.10). This suggests an issue on initialization which quickly balanced the irregular original data brought from input file or created during global queue initialization. In order to main the system in multi-pass calculation, initialization dropping is a serious problem which needs to be handled in next upgrade to maintain the statistical behavior after each global synchronization.

In all, the data in this section from 1D prototype verify the logic design of aCRD system and perform an estimation of calculation time complexity. Meanwhile, these data also simulated several physics features in diffusion system, as the effects from system density and the scale changing relationship between time and space.

4.3. Data from 3D Prototype

In this section, data calculated in 3D system are collected and analyzed to provide a functional verification and to perform more system behavior testing.

In 3D system, the dynamics with coalescence, dissociation and irradiation is applied, which could be described as:

$$(A) + (B) \rightarrow (A+B)$$

The detailed dynamics is designed as Section 4.1 described. All defect clusters are designed to diffuse in a 3-dimension motion in a cubic space. Once multiple defect clusters get close to another, they are assumed to collide with each other and are considered to coalesce. After each major diffusion event, the defect cluster is judged by extra solver to trigger the dissociation event series. Meanwhile, irradiation event series may bring new defects to system with a certain time interval. The data provided by this prototype are designed to verify the function in aCRD algorithm, to run comparison with different scenarios, and to perform evolution simulations.

Fig. 4.12 shows the data from a simulation without irradiation. In this simulation, 1000 single vacancies and 1000 single interstitials are randomly distributed in a cubic space with 500 nm edge length. Overall simulation time for the world clock in aCRD system is 500 seconds. In Fig. 4.12, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (500 s). The y-axis is the relative total vacancies which adds up all existing vacancies in simulation box with their cluster sizes at different time during simulation. Both axes are arranged with log scale.

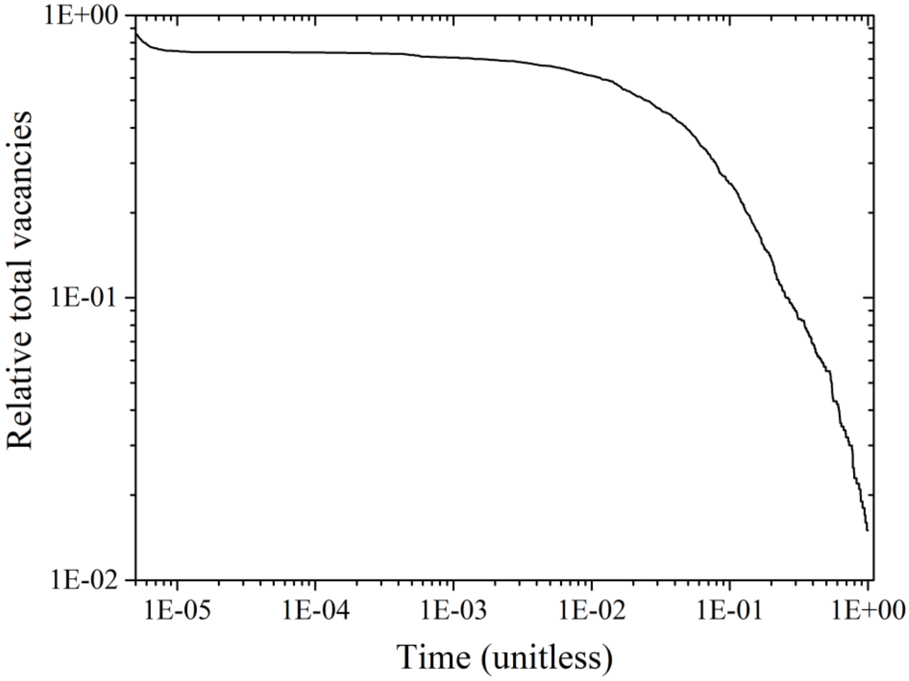


Figure 4.12 aCRD 3D 2000 Defects in Cubic Space with Absorption Boundaries.

For the simulation shown in Fig. 4.12, defect clusters are able to merge with each, the merging of one single vacancy and one single interstitial results in annihilation, while the merging of same types of defects results in cluster buildup. In this way, the dropping of total vacancies is resulted from annihilation and also the transportation of defects to the absorption boundaries.

It is able to tell from Fig. 4.12, 3D system shows the similar behavior compare to 1D system in Fig. 4.9 and Fig. 4.10. The dropping of the system member quantity is able to be separated into several stages for different scales: a quick dropping on defect number during initialization which is the unstable feature from current code systems, a constant dropping period during the change of time scale in mesoscale (in Fig.4.12 from relative time 1E-5 to 1E-2, world clock time from 0.005s to 5s), and a slow dropping period for larger time scale and longer space scale. This trend verified with several basic physical facts, as the system with lower density will trigger the interaction between its members in with longer time scale.

The constant dropping period and the slow dropping period fits with the trend shown in FPKMC and general KMC system [21] . This is regarded to be a positive signal to the framework of code systems. However, as the calculation setup applied in FPKMC is not released with detailed data and parameter library, also considering the early development stage of aCRD system with fixed boundary conditions and setup, current prototype used in aCRD is not able to benchmark with FPKMC system.

As irradiation function is enabled in current aCRD prototypes, more testing calculation is proceeded with irradiation conditions. The following figures show the data from different irradiation conditions. With a configuration of irradiation profile shows in Fig. 4.2 and Fig. 4.13, new defects (A_{irr} sets Frenkel pairs) are randomly added to the simulation system with certain time interval (Δt on simulation world clock).

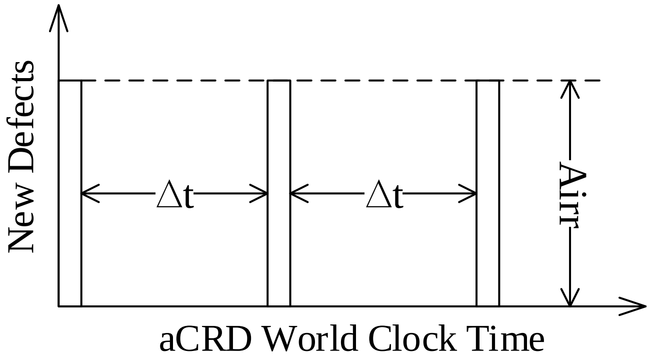


Figure 4.13 aCRD 3D Irradiation Event Profile.

Fig. 4.14 shows the data with A_{irr} = 1 (for each irradiation event, one single vacancy and one single interstitial are introduced to the simulation system) with different irradiation frequencies ((Δt). The data contained in Fig. 4.14 include Δt = 0.05s, 0.04s, 0.035s, 0.0.3s, 0.025s, 0.02s, 0.015s, 0.01s, 0.005s. With comparison and analysis, data from this part demonstrate the irradiation effects with different dose and new defects introducing speed.

Input of these calculation contains 1,000 single vacancies and 1,000 interstitials at random locations inside a cubic space with edge length of 500 nm. Overall simulation time for the world clock in aCRD system is 10 seconds. In Fig. 4.14, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (10 s). The y-axis is the total vacancies which adds up all existing vacancies in simulation space with their cluster sizes at different time during simulation. Both axes are arranged with log scale.

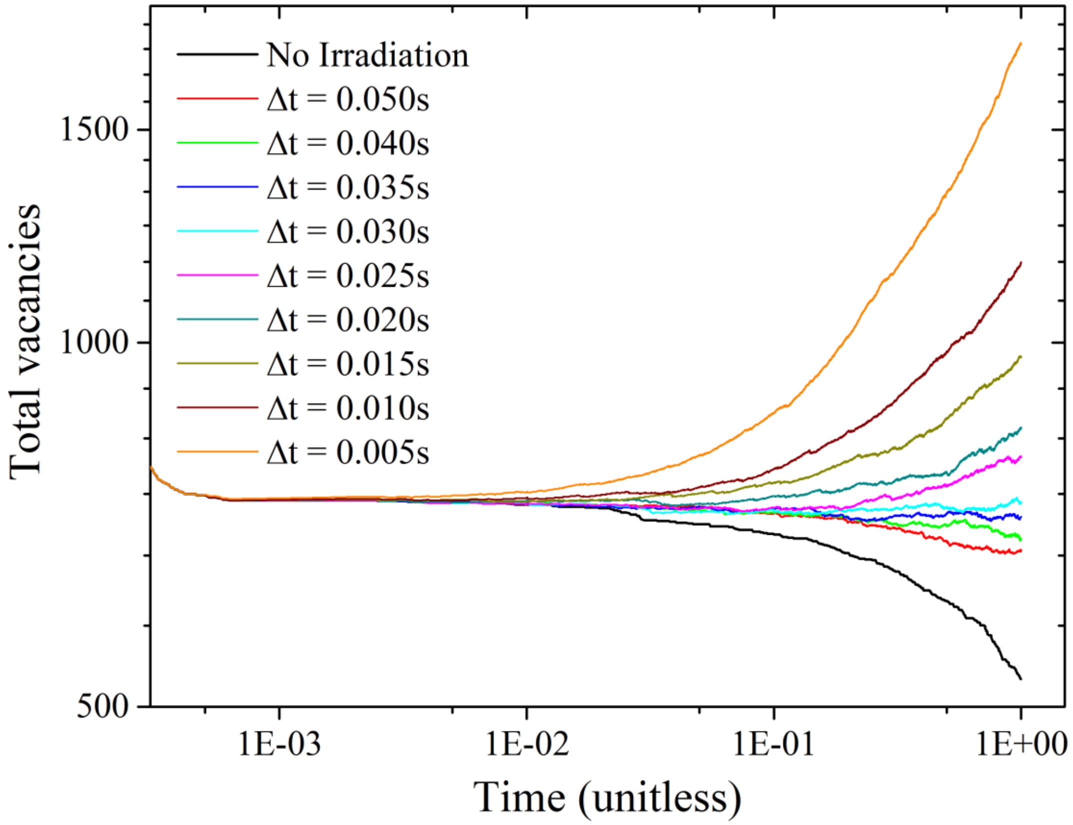


Figure 4.14 aCRD 3D Irradiation Comparison with Different Frequencies.

Comparing with the black curve shown in Fig. 4.14 which represents the trend for defects to annihilate and diffuse out without irradiation, introducing with new defects by irradiation attempts to equilibrate this effect as shown by the cases with lower frequency irradiation (Δt = 0.05s, 0.04s, 0.035s and 0.03s). Once equilibrium between removing from the system and adding to the system is reached, the system begins to accumulate vacancies (Δt = 0.025s, 0.02s, 0.015s, 0.01s and 0.005s).

Limited by the accuracy of current solver and diffusivity database, the current version is still not able to quantify the effects of swelling and to validate with experiment data. However, with the buildup of vacancies being presented as shown in Fig. 4.14, the trend represents a verification of the dynamics and guarantees a positive framework for future developments and upgrades.

A further analysis based on the data obtained from Δt = 0.01s and 0.005s are provided to analyze the accumulation of vacancies. Theoretically, with a factor of 2 on new defects introduced to system, at same world clock stamps, the system with Δt = 0.005s should accumulate double the particles compare to Δt = 0.01s. This comparison is demonstrated in Fig. 4.15. In Fig. 4.15, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (10 s). The y-axis is the accumulated vacancies ratio R_1 calculated with Eq. 4.1:

$$R_1 = (V_{0.005} - V_{No})/(V_{0.01} - V_{No}) \tag{4.1}$$

where $V_{0.005}$ stands for the total vacancies with Δt = 0.005s, $V_{0.01}$ stands for the total vacancies with Δt = 0.01s and V_{No} stands for the total vacancies without irradiation.

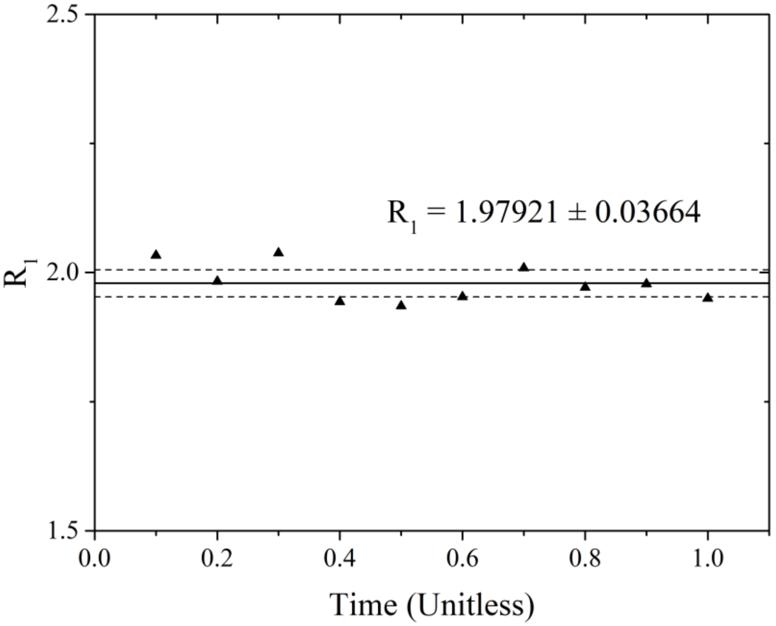


Figure 4.15 aCRD 3D Irradiation Accumulation Ratio Comparison.

With a constant ratio through the simulation time period, Fig. 4.15 shows a very stable behavior on vacancy accumulation in aCRD system. However, for different time stamps, the accumulated vacancies ratio R_1 is not able to be kept with this feature. Fig. 4. 16 shows the data analyzed at different time stamps. With double simulation time, the accumulated vacancies ratio between the curve of Δt = 0.005s and Δt = 0.01s should be kept at 1.0. The value from the current calculated R_1 with the data for Δt = 0.005s at t = 0.4 (unitless) and the data for Δt = 0.01s at t = 0.8 (unitless) is 1.16, which deviates from the assumption value as 1.0. This result suggests a possibility of a latent effect on original vacancy removing speed due to the change of irradiation frequency.

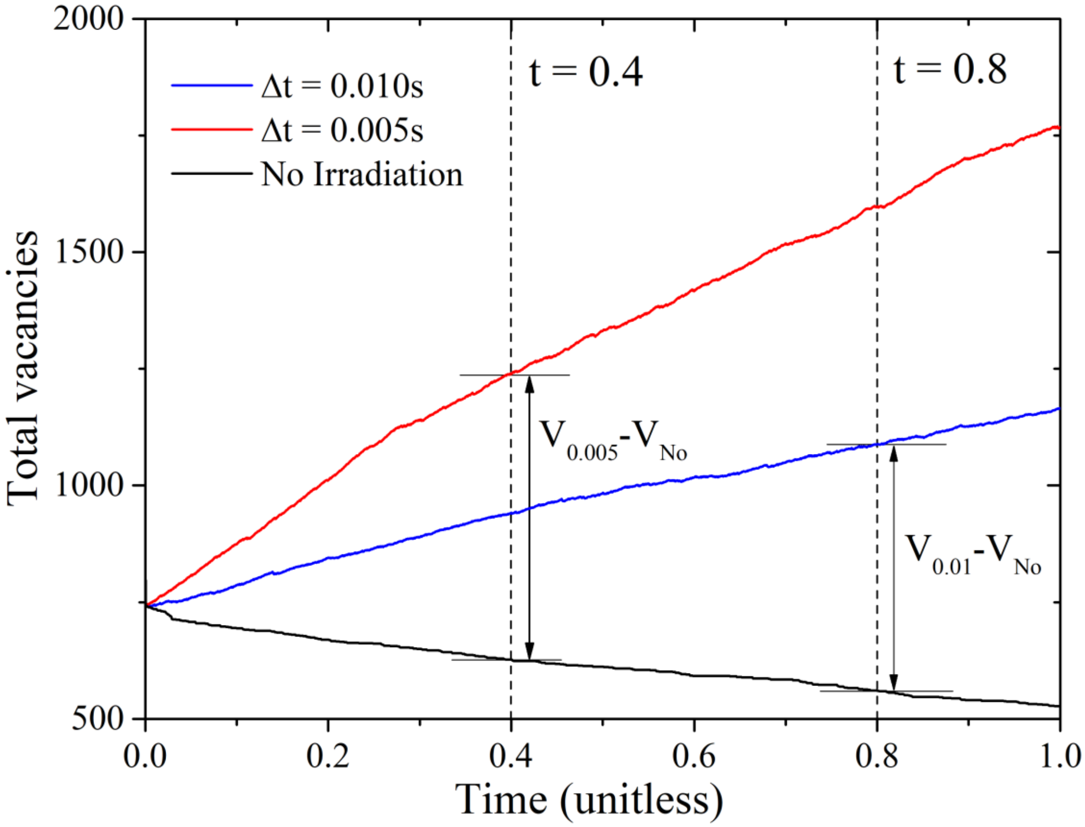
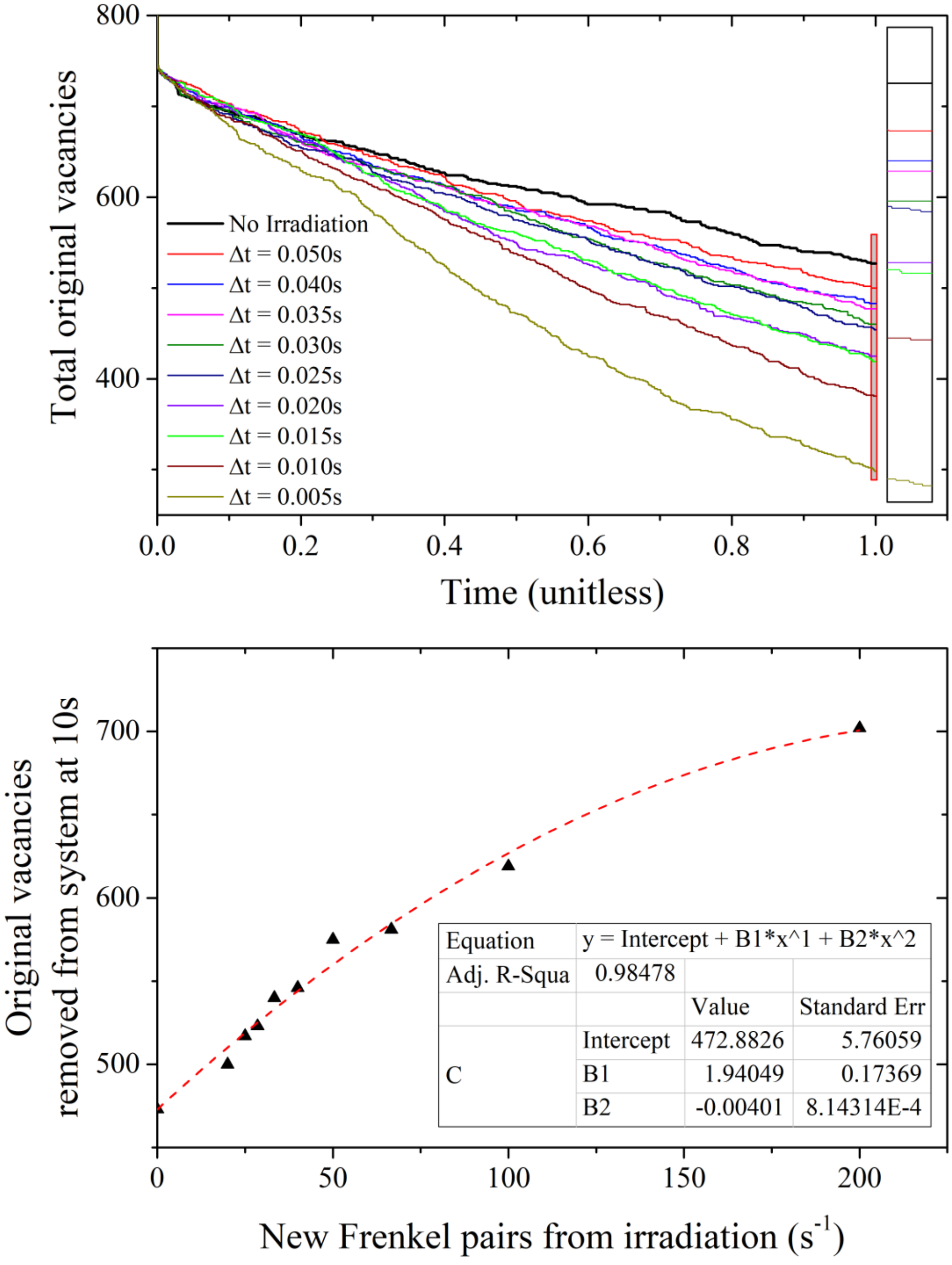


Figure 4.16 aCRD 3D Irradiation Accumulation Ratio at Different Time Stamps.

In order to verify this effects, data shown in Fig. 4.17 are obtained.



Equation	y = Intercept + B1*x^1 + B2*x^2		
Adj. R-Squa	0.98478		
		Value	Standard Err
C	Intercept	472.8826	5.76059
	B1	1.94049	0.17369
	B2	-0.00401	8.14314E-4

Figure 4.17 aCRD 3D Irradiation Analysis on Original Vacancies.

In the first chart of Fig. 4.17, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (10 s) and the y-axis is the total number of existing original vacancies in simulation space during simulation with different irradiation conditions. In the second chart of Fig. 4.17, the x-axis stands for new Frenkel pair introduced to simulation system per second which represents the radiation strength and the y-axis is the total number of original vacancies which have been removed from the system after 10 seconds with different irradiation conditions in aCRD system. All axes are arranged with linear scale.

From the first chart of Fig. 4.17, a trend to promote removing of original defects is revealed by applying irradiation to the system. With irradiations, original defects are removed from the system much quicker comparing to the scenario without irradiation. Also this chart is suggesting that with a higher radiation strength (higher frequency), the quicker original defects are removed from the system.

The second chart of Fig. 4.18 proves the suggestion above. With the analysis between the number of removed original vacancies with 10s simulation in aCRD and radiation strength (new Frenkel pairs introduced to simulation space per second) based on different irradiation frequencies, the positive correlation between these parameters are revealed. As shown in the chart, a second degree polynomial fitting is performed with a high R^2 value. This shows the potential for aCRD to directly estimate the effect of certain phenomenon with proper parameters.

Another important data testing is proceed to study the effect of irradiation frequency with same number of new defects introduced to the system per second.

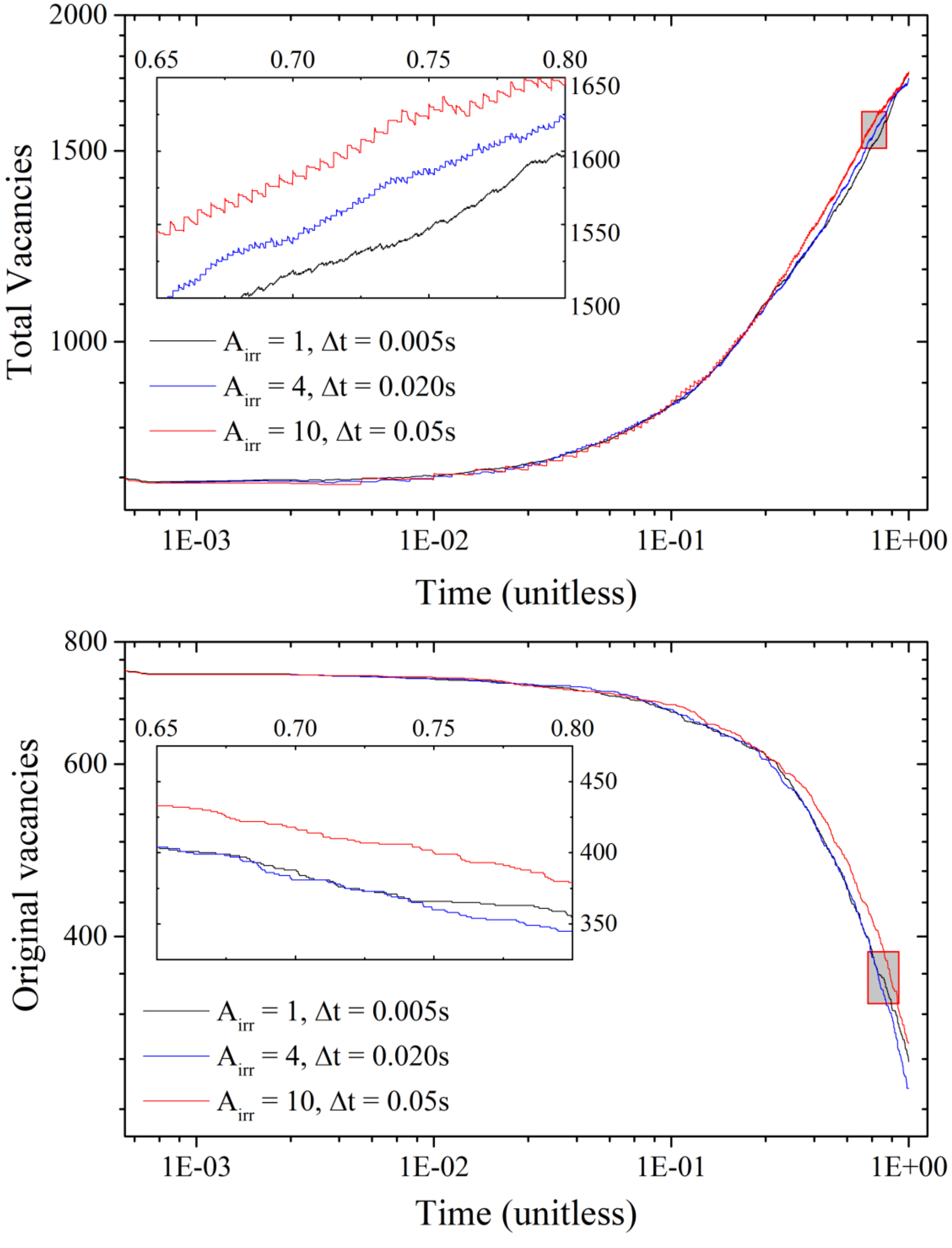


Figure 4.18 aCRD 3D Irradiation Analysis on Irradiation Pulses.

Fig. 4.18 is obtained with the data from the simulations of frequency effects. Refer to Fig. 4.13, different A_{irr} numbers are set to introduce different pairs of Frenkel defects with specific Δt to maintain a strength of introducing 200 Frenkel pairs per second. This part consists of three sets of simulations: 1 pair of Frenkel defect every 0.005s, 4 pairs every 0.02s and 10 pairs every 0.05s.

In the first chart of Fig. 4.18, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (10 s) and the y-axis is the total number of vacancies inside simulation space. In the second chart of Fig. 4.17, the x-axis stands for the world clock in aCRD system arranged to relative time to the wall clock (10 s) and the y-axis is the total number of existing original vacancies in simulation space during simulation with different irradiation conditions. The axes in main charts are arranged with log scale while the axes in enlarged charts are arranged with linear scale.

In the first chart of Fig. 4.18, it is able to tell that three curves with different pulses have different accumulations of vacancies. This chart enlarges the parts during t (unitless) = 0.65 to t (unitless) = 0.80 with a range on y-axis of 150 vacancies. From the enlarged figures, the total vacancies at a specific global clock in the system with an irradiation of 10 Frenkel pairs every 0.05s (red curve) is large than the value with an irradiation of 4 Frenkel pairs every 0.02s, and the total vacancies in the system with an irradiation of 1 pair of Frenkel defect every 0.005s is the smallest one within three sets of data. With the same global clock, the system with higher frequency radiation accumulates less vacancies which suggests a potent suppression of vacancy accumulation for high frequency pulse system.

In order to analyze the bias contributed from the effects on the removing speed of original vacancies, the second chart of Fig. 4.18 is obtained. This chart also enlarges the parts during t (unitless) = 0.65 to t (unitless) = 0.80 with a range on y-axis of 150 vacancies. It is able to tell that the differences contributed in this part are not as significant as the one shown in the first chart of Fig. 4.18. At the same time, the order between different frequencies does not match with the order evaluated by the total vacancies. This is suggesting that this effect may receive more contributions from the interactions between new defects rather than the effects on removing original vacancies.

The significance of differential is calculated with a comparison between the case $\Delta t = 0.05s$ and $\Delta t = 0.005s$. Increment percentage at a specific global time is calculated with Eq. 4.2:

$$\text{Pct} = \left[\left(V_{0.05,\,t} - V_{0.05,\,i}\right) - \left(V_{0.005,\,t} - V_{0.005,\,i}\right)\right] / \left(V_{0.005,\,t} - V_{0.005,\,i}\right) \qquad (4.2)$$

where $V_{0.05,\,t}$ stands for the current total vacancies with $\Delta t = 0.05s$, $V_{0.05,\,i}$ stands for the current total original vacancies with $\Delta t = 0.05s$, $V_{0.005,\,t}$ stands for the current total vacancies with $\Delta t = 0.005s$ and $V_{0.005,\,i}$ stands for the current total original vacancies with $\Delta t = 0.005s$. Table 4.3 records the data within the time range of [0.65, 0.80] which is the parts shown in enlarged chars of Fig. 4.18.

Based on the data from two different radiation condition within a specific time region, the increment percentage between the irradiations with pulse interval of $\Delta t = 0.05s$ and $\Delta t = 0.005s$ is calculated as (4.02 ± 0.79) %. The data presented in this part shows that irradiation with high frequency and lower strength for each pulse might bring a potential effect of suppression of vacancy accumulation. However, as the radiation strength for each pulse is not unified, the temperature effect is also not included in current aCRD, this prototype still doesn't have the capability to simulate a real pulse beam for irradiation research.

Table 4.3 Difference on Increment of Vacancies Resulted from Different Pulses.

Time (unitless)	$V_{0.05, t}$	$V_{0.05, i}$	$V_{0.005, t}$	$V_{0.005, i}$	**Pct**
0.65	1545	433	1463	404	5.00%
0.70	1587	417	1519	388	3.45%
0.75	1632	400	1547	366	4.32%
0.80	1654	379	1589	355	3.32%

The data presented in this section demonstrated the behavior of aCRD prototypes in 3D system with a full dynamics for diffusion and reaction system. These data verified the function design of aCRD system and performed several testing on different scenarios with irradiation. With specific responses on different scenarios, these data from current aCRD framework provided certain evidence on the capability to simulate irradiation conditions for aCRD and provided a strong base for future development.

5. CONCLUSIONS AND FUTURE WORK

In this dissertation, a new multiscale algorithm, the design of its framework of code systems and the code performance data with prototype code are presented for advanced defect cluster dynamics simulation. In this section, summary to current research and development is given and the limitations for current system are examined. Meanwhile, future researches are discussed in order to meet with the expectation to deliver a system which has a valid capability on multiscale simulation for defect cluster dynamics and related phenomena as void swelling.

5.1. Conclusions

Section 2 and Section 3 give a detailed description to the algorithm and the framework of code systems designed for PROJECT aCRD. These designs are regarded to be one of the most important contents in this dissertation. With the designs proposed, an event-driven simulation system is able to be built to simulate the defect cluster dynamics in a wider range of scales both on time and space. Based on the high calculation capability from these designs, the simulation system is able to extend the atomic simulation to mesoscale evolutions, to build a link between microscale simulation and large scale simulation and to finally reach the region where void swelling is simulated with all these atomic level facts.

The prototype code systems as described in Section 4.1 presented the data recorded in Section 4.2 and Section 4.3. These prototypes serve as functional verifications to the design of aCRD system. The 1D data presented in Section 4.2 verified the logic of event-driven dynamics in Section 3 and the capability to describe dynamic events on multiple time scales. The 3D data presented in Section 4.3 verified the function to simulate the full dynamics for defect cluster diffusion and interaction. These data also presented the capability to simulate different scenarios for defect cluster evolution, e.g. evolution with different irradiation frequencies, evolution with different radiation pulses. Meanwhile, the data also reveals several physical features for radiation effects on defect clusters, e.g. the effects that irradiation accelerates the removing of existing defects in a simulation space, the effects that radiations with different pulse change the accumulation of vacancies. These data shows a strong capability of this code system to perform simulation on multiple research topics.

The current code systems also present a good communication between different modules. With current prototypes, upgrades are able to be proceeded to provide the simulation with accuracy on calculation, integrity on kinematics and efficiency on code performance. From there, the aCRD system will be able to validate with experimental data and to reach a higher standard.

5.2. Limitations and Future Work

In the dissertation, numbers of limitation for current prototypes have been spotted. Future work on current aCRD system is focus on overcoming these limitations.

The first limitation is resulted from the assumptions and approximations applied in prototypes. In current prototypes, several dynamics events as dissociation are not able to be simulated with individual events, which makes the communication between defect clusters away from reality. In future upgrade, dynamic events with more real details are necessary, these subroutines are designed to provide more details on the changing of clusters during each events.

Adding to this limitation, the inaccuracy introduced by the prototype solvers also challenges the aCRD system. For all the solvers applied in current prototypes, upgrades are necessary with systematic research. For major diffusion solvers related to the application of Green's function, a systematic research on the sampling method is needed to offer solutions with higher accuracy on exit time, exit location and also the transported location for transient diffusion events. In this part, the effects of temperature and diffusivity changes are necessary to be included. For reaction solver, more researches are necessary to decide a sub-algorithm to run the sampling of final status and reaction time. In this part, the effects of formation energy for different clusters are necessary to be included.

Another limitation is from the neighbor searching method applied in current code systems. With current design, the neighbor searching method takes a time complexity of O (N^2), which lowers down the efficiency during the calculation with large number. Though the first-passage method and asynchronous event-driven [94] design accelerate the algorithm, the limitation on searching method decelerates the code systems. The application of neighbor listing [95] [96] or advanced searching [97] method is optional for future development, which also related to the improvement of space allocation method.

For current aCRD prototypes, the simulation of 1D/3D hybrid diffusion in 3D space is still not allowed, which is a technique challenge on space allocation module. This function is regraded to be critical to simulate swelling incubation period. In future development, a separated space allocation module is in need to assign the simulation system with high flexibility and capability on multiple dimension assignment. In this part, the upgrade of spatial information storage is also on schedule, which may also offer the solution to the challenge on searching method. At the same time, flexible boundary conditions are also optional for space allocation module. With this function, the system will be able to simulate the interactions without the effects of absorption from boundaries which quickly removes all existing defects.

Meanwhile, the current aCRD system is not capable for parallel calculation which makes it not capable on supercomputing architecture. This feature also limits the advantage from the design of separated solver interfaces as described in Section 3.2.2. Also, for realistic calculation comparing with experiment, large scale calculations are required with CPU hours which cannot be proceeded without parallel processing. In future work, a parallel calculation upgrade should also be on the schedule.

REFERENCES

[1] L. K. Mansur, “Void Swelling in Metals and Alloys under Irradiation: An Assessment of the Theory”, Nuclear Technology, **40:1**, 5-34, 1978.

[2] E. A. Little, “Void-Swelling in Irons and Ferritic Steels I. Mechanisms of swelling suppression”, J. Nucl. Mat., **87**, 11-24, 1979.

[3] F. Takeyama *et al*., *Fundamental Aspects of Radiation Damage in Metals*, **2**, Natl. Tech. Information Service, 1100, 1976.

[4] F. Garner *et al*., “Recent Developments Concerning Potential Void Swelling of PWR Internals Constructed from Austenitic Stainless Steels”, Contribution of Materials Investigation to the Resolution of Problems Encountered in Pressurized Water Reactors, 22, 2002.

[5] G. Bond *et al*., “Void Swelling of Annealed 304 Stainless Steel at ~370-385°C and PWR-Relevant Displacement Rates”, 9th International Conference on Environmental Degradation of Materials in Nuclear Power Systems – Water Reactors, 1045-1050, 1999.

[6] F. Garner *et al*., "Neutron-induced swelling of commercial alloys at very high exposures", Effects of Radiation on Materials: 14th International Symposium (Volume II), ASTM International, 1990.

[7] E. A. Little, Radiat. Eff., **16**, 135, 1972.

[8] J. J. Huet *et al*., Nucl. Technol., **24**, 216, 1974.

[9] D. R. Arkell *et al*., J. Nucl. Mat., **74**, 114, 1978.

[10] F. A. Smidt *et al*., *Defects and Defect Clusters in BCC Metals and their Alloys*, Ed. R.J. Arsenault, 341, 1973.

[11] W. G. Johnston *et al*., “An experimental survey of swelling in commercial Fe-Cr-Ni alloys bombarded with 5 MeV Ni Ions”, J. Nucl. Mat., **54**, 24, 1974.

[12] F. A. Smidt *et al*., ASTM Special Tech. Publ., **611**, 227, 1976.

[13] K. Farrell *et al*., J. Nucl. Mat., **35**, 352, 1970.

[14] G. L. Kulcinski *et al*., Radiat. Eff., **2**, 57, 1969.

[15] D. Raabe *et al.*, Max-Planck-Gesellschaft, www.mpg.de/19242/Multiscale_modelling.

[16] J. B. Anderson, "Quantum chemistry by random walk. H ^2P, H^+_3 D_{3h} $^1A'_1$, H_2 $^3\sum^+_u$, H_4 $^1\sum^+_g$, Be ^1S", J. Chem. Phys. **65**, 4121, 1976.

[17] A. B. Bortz *et al.*, "A New Algorithm for Monte Carlo Simulation of Ising Spin Systems", J. Comput Phys, **17**, 10, 1975.

[18] N. Metropolis *et al.*, "Equation of State Calculations by Fast Computing Machines", J. Chem. Phys., **21**, 1087, 1953.

[19] F. M. Bulnes *et al.*, "Collective surface diffusion: n-fold way kinetic Monte Carlo simulation", Phys. Rev. E, **58**, 86, 1998.

[20] S. K. Theiss *et al.*, "Atomic scale models of ion implantation and dopant diffusion in silicon", Thin Solid Films, **365**, 219, 2000.

[21] T. Oppelstrup *et al.*, "First-Passage Monte Carlo Algorithm: Diffusion without All the Hops", Phys. Rev. Lett., **97**, 230602, 2006.

[22] R. E. Stoller *et al.*, "Mean field rate theory and object kinetic Monte Carlo: A comparison of kinetic models", J. Nucl. Mat., **382**, 77-90, 2008.

[23] S. D. Harkness *et al.*, "A study of void formation in fast neutron-irradiated metals", Metall. Trans., **2**, 1457, 1971.

[24] H. Wiedersich, "On the theory of void formation during irradiation", Radiat. Eff., **12**, 111, 1972.

[25] A. D. Bralisford *et al.*, "The rate theory of swelling due to void growth in irradiated metals", J. Nucl. Mat., **44**, 121, 1972.

[26] L. K. Mansur *et al.*, *Int. Conf.*, "Defects and Defect Clusters in BCC Metals and Their Alloys", August 14-16, 1973.

[27] L. K. Mansur *et al.*, "Properties of Reactor Structural Alloys after Neutron or Particle Irradiation," ASTM-STP-570, 272, 1975.

[28] W. G. Wolfer, "Fundamental Aspects of Radiation Damage in Metals", CONF-751006-P2, Vol. II, 812, 1975.

[29] M. H. Yoo *et al.*, "Steady-state diffusion of point defects in the interaction force field", Phys. Status Solidi B, **77**, 181, 1976.

[30] R. W. Baluffi, “Fundamental Aspects of Radiation Damage in Metals”, CONF-751006-P2, Vol. II, 852, 1975.

[31] P. T. Heald *et al*., Acta Metall., **23**, 1389, 1975.

[32] A. D. Brailsford *et al*., *Proc. Conf. Vacancies*, 76, University of Bristol, 1976.

[33] A. D. Brailsford *et al*., “Point defect sink strengths and void-swelling”, J. Nucl. Mat., **60**, 246, 1976.

[34] A. D. Brailsford *et al*., “Physical Metallurgy of Reactor Fuel Elements”, September 2-7, 1973.

[35] G. R. Odette *et al*., *Prac. Int. Conf.*, “Radiation Effects and Tritium Technology for Fusion Reactors”, October 1-3, 1975, CONF-750989, Vol. I, 395, 1976.

[36] J. R. Beeler, Jr., "Computer Experiments on Radiation-Induced Defect Production and Defect Annealing", Trans. Am Nucl. Soc., **27**, 314, 1977.

[37] A. J. E. Foreman, Radiat. Eff., **21**, 81, 1974.

[38] J. L. Straalsund, J. Nucl. Mat., **51**, 302, 1974.

[39] R. Bullough *et al*., AERE R7952, U.K. Atomic Energy Research EstabHshment, Harwell, Oxfordshire, Feb. 1975.

[40] D. G. Doran *et al*., *Prac. Workshop*, “Correlation of Neutrons and Charged Particle Damage”, Oak Ridge National Laboratory, June 8-9, 1976, CONF-760673, 3, U.S. Energy Research and Development Administration, 1976.

[41] R. Bullough *et al*., "Irradiation Effects on Structural Alloys for Nuclear Reactor Apphcations", ASTM-STP-484, 317, American Society for Testing and Materials, 1970.

[42] K. Urban *et al*., “Growth of Defect Clusters in Thin Nickel Foils during Electron Irradiation. II. Temperature Dependence of the Growth Rate of Interstitial Loops”, Phys. Status Solidi A, **6**, 173, 1971.

[43] A. J. E. Foreman, Radiat. Eff., **14**. 175, 1972.

[44] N. Q. Lam *et al*., “Steady-state point-defect diffusion profiles in solids during irradiation”, Radiat. Eff., **23**, 53, 1974.

[45] G. L. Guthrie *et al*., Scripta Metall., **9**, 1149, 1975.

[46] C. J. Saving, AERE TP-610, U.K. Atomic Energy Research Establishment, Harwell, Oxfordshire, July 1975.

[47] M. H. Yoo *et al*., “Distributions of point defects in bounded media under irradiation”, J. Nucl. Mat., **62**, 282, 1976.

[48] A. D. Brailsford *et al*., “Effect of self-ion injection in simulation studies of void swelling”, J. Nucl. Mat., **71**, 110, 1977.

[49] M. H. Yoo *et al*., "General Rate Theory Model of Void Swelling in Irradiated Metals", Trans. Am. Nucl. Soc., **27**, 326, 1977.

[50] A. D. Brailsford *et al*., “An effect of solute segregation on void growth in irradiated dilute alloys”, J. Nucl. Mat., **56**, 7, 1975.

[51] P. R. Okamoto *et al*., J. Nucl. Mat., **53**, 336, 1974.

[52] F. V. Nolfi Jr., “Elastic interactions between voids induced by solute segregation”, J. Appl. Phys., **47**, 24, 1976.

[53] L. K. Mansur *et al*., “Influence of a surface coating on void formation”, J. Nucl. Mat., **69-70**, 825, 1978.

[54] R. A. Johnson *et al*., Phys. Rev. B, **13**, *10*, 4364, 1976.

[55] F. A. Smidt Jr. *et al*., Scripta Metall., **7**, 495, 1973.

[56] J. S. Koehler, “Decrease in the void growth rate by interstitial trapping”, J. Appl. Phys., **46**, 2423, 1975.

[57] W. Schilling *et al*., *Consultant Symp.* “Physics of Irradiation Produced Voids”, September 9-11, 1974, Harwell, England, AERE-R 7934, 212, U.K. Atomic Energy Research Establishment, 1974.

[58] P. R. Okamoto *et al*., *Proc. Workshop*, “Correlation of Neutrons and Charged Particle Damage”, Oak Ridge National Laboratory, June 8-9, 1976, CONF-760673, 111, U.S. Energy Research and Development Administration, 1976.

[59] A. D. Brailsford *et al*., *Proc. Int. Conf*, “Properties of Atomic Defects in Metals”, October 18-22, 1976, Argonne National Laboratory, J. Nucl. Mat., **69-70**, 434, 1978.

[60] H. Wiedersich *et al*., *Proc. Workshop*, “Correlation of Neutron and Charged Particle Damage”, Oak Ridge National Laboratory, June 8-9, 1976, CONF-760673, 21, U.S. Atomic Energy Research and Development Administration, 1976.

[61] P. T. Heald, *Proc. Int. Conf.*, “Radiation Effects in Breeder Reactor Structural Materials”, June 19-23, 1977, 781, M. L. BLEIBERG and J. W. BENNETT, Eds., American Institute Mining, Metallurgical and Petroleum Engineers, 1977.

[62] W. G. Wolfer *et al.*, *Proc. Int. Conf.*, “Radiation Effects in Breeder Reactor Structural Materials”, June 19-23, 1977, 841, M. L. BLEIBERG and J. W. BENNETT, Eds., American Institute Mining, Metallurgical and Petroleum Engineers, 1977,

[63] D. G. Doran *et al.*, *Proc. Int. Conf.*, “Radiation Effects in Breeder Reactor Structural Materials”, June 19-23, 1977, 591, M. L. BLEIBERG and J. W. BENNETT, Eds., American Institute Mining, Metallurgical and Petroleum Engineers, 1977.

[64] L. K. Mansur, *Kinetics of Nonhomogeneous Processes*, Wiley-Interscience, New York, p. 377, 1987.

[65] M. Kiritani, “Analysis of the Clustering Process of Supersaturated Lattice Vacancies”, J. Phys. Soc. Jpn., **35**, 95, 1973.

[66] J. L. Katz *et al.*, “Effect of insoluble gas molecules on nucleation of voids in materials supersaturated with both vacancies and interstitials”, J. Nucl. Mat., **46**, 41, 1973.

[67] N.M. Ghoniem *et al.*, “A numerical solution to the fokker-planck equation describing the evolution of the interstitial loop microstructure during irradiation”, J. Nucl. Mat., **92**, 121, 1980.

[68] S.I. Golubov *et al.*, “Grouping method for the approximate solution of a kinetic equation describing the evolution of point-defect clusters”, Philos. Mag. A, **81**, 643, 2001.

[69] J. Marian *et al.*, “Stochastic cluster dynamics method for simulations of multispecies irradiation damage accumulation”, J. Nucl. Mat., **415**, 84, 2011.

[70] R. E. Rudd *et al.*, “Coarse-grained molecular dynamics: Nonlinear finite elements and finite temperature”, Phys. Rev. B, **72**, 144104, 2005.

[71] L. Chen, “Phase-Field Models for Microstructure Evolution”, Annual Review of Materials Research, **16**, 113, 2002.

[72] M. Plapp *et al.*, “Multiscale Finite-Difference-Diffusion-Monte-Carlo Method for Simulating Dendritic Solidification”, Phys. Rev. E, **60**, 6865, 1999.

[73] S. Redner, *A Guide to First-Passage Processes*, Cambridge University Press, Cambridge, 2001.

[74] M. H. Kalos *et al*., “Helium at zero temperature with hard-sphere and other forces”, Phys. Rev. A, **9**, 2178, 1974.

[75] J. S. Van Zon *et al*., “Simulating Biochemical Networks at the Particle Level and in Time and Space: Green's Function Reaction Dynamics”, Phys Rev Lett, **94**, 128103, 2005.

[76] Takahashi K *et al*., “Spatio-temporal correlations can drastically change the response of a MAPK pathway”, Proc. Natl Acad Sci USA, **107**, 2473, 2010.

[77] Sokolowski TR *et al*., “Spatial-Stochastic Simulation of Reaction-Diffusion Systems”, arXiv, 2017.

[78] T. Oppelstrup *et al*., “First-Passage Kinetic Monte Carlo method”, arXiv, 2009.

[79] A. Donev *et al*., “A First-Passage Kinetic Monte Carlo Algorithm for Complex Diffusion-Reaction Systems”, J. Comp. Phys., **229**, 3214, 2010.

[80] A. Vijaykumar *et al*., “Combining molecular dynamics with mesoscopic Green's function reaction dynamics simulations”, J. Chem. Phys., **143**, 214102, 2015.

[81] D. Zhong *et al*., “Large-scale simulations of diffusion-limited n-species annihilation”, Phys. Rev. E, **67**, 040101(R), 2003.

[82] D. ben-Avraham, “Computer simulation methods for diffusion-controlled reactions”, J. Chem. Phys., **88**, 941, 1988.

[83] J. S. Van Zon *et al*., “Green’s-function reaction dynamics: A particle-based approach for simulating biochemical networks in time and space”, J. Chem. Phys., **123**, 234910, 2005.

[84] T. R. Sokolowski *et al*., "eGFRD in all dimensions”, J. Chem. Phys., **150**, 054108, 2019.

[85] C.-C. Fu *et al*., “Multiscale modelling of defect kinetics in irradiated iron”, Nat. Mater. **4**, 68, 2005.

[86] J. A. Given *et al*., “A first-passage algorithm for the hydrodynamic friction and diffusion-limited reaction rate of macromolecules”, J. Chem. Phys. **106**, 3761, 1997.

[87] M. Mascagni *et al*., “Monte Carlo methods for calculating some physical properties of large molecules”, SIAM J. Sci. Comput. **26**, 339, 2004.

[88] D. M. Ceperley *et al.*, in *Monte Carlo Methods in Statistical Physics*, edited by K. Binder (Springer-Verlag, Berlin), 1979.

[89] M. Smith *et al.*, “Constant-number Monte Carlo simulation of population balances”, Chem. Eng. Sci. **53**, 1777, 1998.

[90] K. Zuse, *Der Plankalkül*, Konrad Zuse Internet Archive. 2.47–2.56, 1972.

[91] E. F. Moore, *The shortest path through a maze*, Proceedings of the International Symposium on the Theory of Switching, Harvard University Press. 285–292, 1959.

[92] CY Lee, “An Algorithm for Path Connections and Its Applications”, IRE Transactions on Electronic Computers, **EC-10**, 1961.

[93] S. S. Skiena, “Sorting and Searching”, *The Algorithm Design Manual*, 103-144, Springer, London, 2008.

[94] A. Donev, “Asynchronous event-driven particle algorithms”, SIMULATION: Transactions of The Society for Modeling and Simulation International, **85**, 229–242, 2008.

[95] L. Verlet, “Computer ‘experiments’ on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules”, Phys. Rev., **159**, 98–103, 1967.

[96] B. Quentrec *et al.*, “New method for searching for neighbors in molecular dynamics computations”, J. Comp. Phys., **13**, 430-432, 1973.

[97] J. L. Bentley, “Multidimensional binary search trees used for associative searching”, Communications of the ACM, **18**, 5091, 1975.

APPENDIX A

PROJECT ACRD PROTOTYPE CODE SYSTEMS CHANGE LOG

```
## 1.0.4 – 2019-02-05

### Changed

    -   Diffusivity library [Library Module] values.

### Fixed

    -   Several minor running errors in event loop [Event Module].


## 1.0.3 – 2019-01-28

### Added

    -   LAMMPS style dumping procedure [System Module] to visualize data.

### Fixed

    -   Several running errors in event loop [Event Module].


## 1.0.2 – 2019-01-27

### Added

    -   Multiple dumping procedures [System Module] to collect related data.

### Fixed

    -   Several running errors in event loop [Event Module].

    -   Several running errors in main stage control [System Module].
```

## 1.0.1 – 2019-01-27

### Fixed

- A minor error on time sampling function in major diffusion solver [Solver Module].

## 1.0.0 – 2019-01-26

### Added

- Dissociation solver module [Solver Module].

### Fixed

- Several running errors in event loop [Event Module].

## 0.5.1 – 2019-01-26

### Added

- A demonstration event series on dissociation event series [Event Module].

### Fixed

- Several running errors in event loop [Event Module].

## 0.4.2 – 2019-01-26

### Fixed

- A critical error on time sampling function in major diffusion solver [Solver Module].

## 0.4.1 – 2019-01-25

### Added

- New walker event series [Event Module] to simulate the irradiation scenario.

## 0.3.1 – 2019-01-24

### Added

- 3D coalescence event series [Event Module].
- Local reaction solver module [Solver Module].

### Changed

- Boundary setting [System Module] to perform an absorbing boundary.

### Fixed

- A critical error on EVENT_POINTER [System Module].
- Several running errors in event loop [Event Module].

## 0.2.1 – 2019-01-23

### Added

- 3D diffusion and annihilation event series [Event Module].

### Changed

- Boundary setting [System Module] to perform an absorbing boundary.
- Library for diffusivities [Library Module] to describe larger number of species.

### Fixed

- Several running errors in event loop [Event Module].

## 0.1.1 – 2019-01-15

### Added

- ASMS procedure [System Module].

### Changed

- General parameter setting [System Module] to arrange 3D information storage.
- Green’s function solver module [Solver Module] to run sampling with better logic with rejections.

### Fixed

- Several running errors in event loop and tool functions [Event Module].

## 0.1.0 – 2018-11-11

### Added

- Green’s function solver module [Solver Module].
- Tool functions to handle math calculation in aCRD [System Module].
- 1D diffusion and annihilation event series [Event Module].

### Changed

- General parameter setting [System Module] to storage walker information in better style.

### Fixed

- Several running errors in main stage control loop [System Module].

## 0.0.1 – 2018-09-21

### Added

- This change log file to serve as a reference to future developers on PROJECT aCRD.
- Main stage control loop [System Module].
- General parameters setting [System Module].
- Global event queue [System Module].
- Library for diffusivities [Library Module].

APPENDIX B

PROJECT ACRD PROTOTYPE CODE SYSTEMS MODULE LIST

The coming tables contain the functions and short introductions to each.

Table B.1 Table of Functions in aCRD Prototype Code Systems.

Module and Location	Function	Instruction
System Module in toolbox	releaseModule	Release the dynamic-link libraries.
	getDistance	Calculate the center distance within two walkers.
	getSoftDomainASurface Distance	Calculate the surface distance between a walker and the closer center walker of a type I SPD.
	getSoftDomainACenter toCenterDistance	Calculate the center distance between a walker and the closer center walker of a type I SPD.
	getEstimateRadius	Estimate the radius for defect clusters based on sphere assumption.
	getHardDomainCenter Distance	Calculate the distance between the centers of two HPDs.
	getHardDomainSurface Distance	Calculate the distance between the surfaces of two HPDs.
System Module in main stage	checkWalkerOverlapping	Check whether walkers overlap with each other, if so, disable one walker.
	initialSoftDomainA	Build a new SPD.
	initialUpdateSoft DomainA	Add a new member and shift the center of a SPD during initialization.

Table B.1 (continued).

System Module in main stage	checkInitialSoftDomains	Check surface distance between walkers, if one is very close to another, build a new SPD.
	initialHardDomain	Build a new HPD.
	setInitialHardDomains	Do space allocation to set new HPDs.
	getHardDomainModifier	Modify HPDs while they overlap with each other.
	checkInitialDomain	Check all unprotected walkers and initialize the space based on partial ASMS logic.
Library Module	getDvValue	Search for the related diffusivity for a cluster from library.
Solver Module in E01 solver	getMajorDiffTimeStamp	Calculate the event time of E01 event in a HPD.
	getMajorDiffRelativeLocation	Calculate the final location of E01 event in a HPD.
Solver Module in E02 solver	getTransDiffRelativeLocation	Calculate the shift location of E02 event during the break of a HPD
Solver Module in E10 solver	getReactionTimeStamp	Calculate the event time of E10 in a SPD
	getReactionFinalStatus	Calculate the final members of E10 in a SPD
Solver Module in E11 solver	getDissociationFlag	Calculate whether the walker will dissociate during current event.

APPENDIX C

PROJECT ACRD PROTOTYPE SOURCE CODE: MAIN

The following content provides part of the C++ source code used in Section 4. Limited by the content size, notes are not fully included in this appendix.

The code attached in this appendix is regarded to be an effective example of the calculation system. As the source code is designed to be released with GNU Public License, the final version for aCRD system should be referred to the released code with official license.

File: main.h

```
#pragma once
// Version 1 alpha demo
#ifndef MAIN_H
#define MAIN_H

#include <stdio.h>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string>
#include <fstream>
#include <map>
#include <vector>
#include <queue>
#include <iomanip>
#include <Windows.h>
using namespace std;

#define MAJOR_VERSION_NUMBER 1
#define MINOR_VERSION_NUMBER 0
#define REVISIONB_NUMBER 4
```

```
#define VERSION_ID "Alpha"
#definePI 3.141592653589793238
#define eps 1e-16
#define EXTRA_LARGE 4999.99

struct location {
        double x;
        double y;
        double z;
};
struct domaincenter {
        double x;
        double y;
        double z;
        double r;
};
struct walker {
        long ID;
        int type; // 0 V_Based; 1 I_Based; -1 for Not Using
        long component;
        location center;
        double radius;
        double energy; // In keV
        double diffusivity; // In nm^2/s
        int statue; // 0 Released; 1 Soft Protected; 2 Hard Protected
        long regionID;
        double timestamp;
};
struct dissociation_final_walker {walker member[5];};
struct reaction_current_walker {walker member[50];};
struct SPDb_current_walker {walker member[50];};
struct reaction_final_walker {walker member[5];};
struct hard_domain {
        long ID;
        double clock; // Inner timestamp
        long walker_ID;
        int walker_type;
        double radius;
        location center;
        int events_type; // Type 01 or Type 11, -1 for no event (disable)
        double events_time; // Event timestamp
        int event_card; // To the index of event card in EVENT_LIST
        location e01_final_location;
        dissociation_final_walker e11_final_statue;
};
struct soft_domain_a {
        long ID;
        double clock;
        int current_member;
        int final_member;
        reaction_current_walker member_list;
        domaincenter center1;
        domaincenter center2;
        double center_distance; // Center walkers surface distance, always less than
THRES_2
        int events_type; // Type 10, -1 for no event (Disable)
        double events_time; // Event timestamp, relative to the initial time of domain
        int event_card; // To the index of event card in EVENT_LIST
        reaction_final_walker e10_final_statue; // Walker ID will be assigned only when
they are released};
struct soft_domain_b {
        long ID;
        double clock;
        int current_member;
        SPDb_current_walker member_list;
        double radius;
        location center;
        int events_type; // Type 10 or Type 20+, -1 for no event (Disable)
        double timestep; // Hybrid factor, can be global, unit in s};
struct event_card {
        long domainID;
        int domainType; // 0 for HPD, 1 for SPDa, 2 for SPDb
        int eventType; // Only for New Walker, 0 for Domain Event, 20 for New Walker
        double timestamp; // This is world clock stamp, not the relative stamp for
solvers};
event_card EVENT_LIST[200000];
long EVENT_INDEX = 0; // For initialization
long EVENT_POINTER = 1; // For event execulation
long EVENTS_LENGTH = 0;
long S0_INDEX = 1;
long S0_EVENT[5000];
int S0_FULL_FLAG;
#defineINPUTS_SIZE 20000
#define DOMAIN_SIZE 100000
walker ORI_WALKERS[INPUTS_SIZE];
hard_domain HARD_PD[DOMAIN_SIZE];
soft_domain_a SOFT_PD_A[DOMAIN_SIZE];
long i, j, k;
```

```
int inputNumber = 0;
int space_dimension = 0;
int irradiationCheck = 0;
double irradiationPulse = 0.0;
int irradiationINumber = 0;
int irradiationVNumber = 0;
double new_walker_distance = 0.0;
double X_LOWWER_LIMIT = 0.0;
double X_UPPER_LIMIT = 0.0;
double Y_LOWWER_LIMIT = 0.0;
double Y_UPPER_LIMIT = 0.0;
double Z_LOWWER_LIMIT = 0.0;
double Z_UPPER_LIMIT = 0.0;
double WORLD_CLOCK = 0.0; // Point to the last exist S0 clock
double END_CLOCK = 1000.00; // Unit in s
double EXTRA_CLOCK = 0.0; // Add to END_CLOCK
long TIME_STEP = 0;
long WALKER_INDEX = 0; // Point to the last one, storage from 1 to N
long HPDOMAIN_INDEX = 0; // Point to the last one, storage from 1 to N
long SPDOMAINA_INDEX = 0; // Point to the last one, stotage from 1 to N
long SPDOMAINB_INDEX = 0;
long WALKER_COUNTER = 0;
long HPD_COUNTER = 0;
long SPDA_COUNTER = 0;
long SPDB_COUNTER = 0;
long DUMP_COUNTER = 0;
double RANDOM_SEED = 0.0;
double TEMP_TIME = 0.0;
int CONTROL_FLAG;
int STAGE_FLAG;
int INPUT_FLAG;
int LOCATE_FLAG;
long NEARID;
double NEARDISTANCE = 0.0;
string filenameInput;
string OUTPUTNAME;
string STAMPNAME;
int DISPLAYFLAG_SYS = 1;
int DISPLAYFLAG_DEB = 0;
int DUMPFLAG_INN = 1;
int DUMPLAMMPSFLAG_INN = 1;
int DUMPCOUNTER_FLAG = 1;
int DUMPCOUNTER_INN = 0;
string ININOTES;
```

```
void releaseModule();
double getDistance(walker WALKER_1, walker WALKER_2);
double getDistance1D(walker WALKER_1, walker WALKER_2);
double getSoftDomainASurfaceDistance(walker WALKER_NEW, soft_domain_a
SPDA);
double getSoftDomainASurfaceDistance1D(walker WALKER_NEW, soft_domain_a
SPDA);
double getSoftDomainACentertoCenterDistance(walker WALKER_NEW,
soft_domain_a SPDA);
double getSoftDomainACentertoCenterDistance1D(walker WALKER_NEW,
soft_domain_a SPDA);
double getHardDomainCenterDistance(hard_domain DOMAIN_1, hard_domain
DOMAIN_2);
double getEstimateRadius(walker WALKER);
long getNeighborDomainID(location DOMAIN_CENTER);
double getHardDomainSurfaceDistance1D(walker WALKER, hard_domain HPD);
double getHardDomainSurfaceDistance3D(walker WALKER, hard_domain HPD);
void checkWalkerOverlapping();
void initialSoftDomainA3D(long ID, walker WALKERS_1, walker WALKERS_2);
void initialUpdateSoftDomainA3D(long ID, walker WALKERS_NEW);
void checkInitialSoftDomains3D();
void initialSoftDomainA1D(long ID, walker WALKERS_1, walker WALKERS_2);
void initialUpdateSoftDomainA1D(long ID, walker WALKERS_NEW);
void checkInitialSoftDomains1D();
void initialHardDomain3D(long ID, walker WALKERS, double DOMAIN_R);
void setInitialHardDomains3D();
void initialHardDomain1D(long ID, walker WALKERS, double DOMAIN_R);
void setInitialHardDomains1D();
double getHardDomainModifier(hard_domain DOMAIN_1, hard_domain DOMAIN_2);
void checkInitialDomain1D();
void checkInitialDomain3D();
int recheckInitialDomain();
void buildHardDomain(long ID, walker WALKERS);
void buildSoftDomainA(long ID, walker WALKERS_1, walker WALKERS_2);
void updateSoftDomainA(long ID, walker NEW_WALKER);
HMODULE dllLibDiffusivity;
HMODULE dllSolverMajorDiff;
HMODULE dllSolverTransientDiff;
HMODULE dllSolverLocal;
HMODULE dllSolverDissociation;
#endif
```

File: Parameter.h

```
#pragma once
// Version 1 alpha demo
// Further dll Prototype for parameters
// The main input memebers use the following units
// Atomic radius : nm
// Crystal structure : nm
// Diffusivity: m^2/s - nm^2/s
// Time : s
// Energy : keV

#ifndef PARAMETER_H
#define PARAMETER_H

#define THRES_1 1.00 // Trigger of E02 during protecting, with another HPD
#define THRES_2 0.50 // Trigger of E10 during protecting, with another walker. 4 x single radius
#define THRES_3 1.50 // Minimum distance in E11 releasing, within final walkers
// Unit in s
#define LIMIT_1 5.0e-8 // E10 Holdup destination time, stamp for buildstamp + LIMIT_1

#define INITIAL_EVENTS_TIME 0.001
#define FE_RADIUS 0.126
#define FE_BCC_A 0.28665 // Assume to be single vacancy
#define INTERST_MODIFIER 1.00
#define VACANCY_MODIFIER 1.25
#define INIDOMAIN_MODIFIER 6.0 // for sqrt (Factor * Dv * t).

#endif
```

File: Toolbox.h

```
#pragma once

#ifndef TOOLBOX_H
#define TOOLBOX_H

void releaseModule(){
	cout << "Release Module from System." << endl;
	if (dllSolverDissociation != NULL)   {FreeLibrary(dllSolverDissociation); }
	if (dllSolverLocal != NULL)  {FreeLibrary(dllSolverLocal);}
	if (dllSolverTransientDiff != NULL) {FreeLibrary(dllSolverTransientDiff);}
	if (dllSolverMajorDiff != NULL) {FreeLibrary(dllSolverMajorDiff);}
	if (dllLibDiffusivity != NULL) {FreeLibrary(dllLibDiffusivity);}
	cout << "Have released all Modules." << endl;
}
double getDistance(walker WALKER_1, walker WALKER_2) {
	double l = 0.0;
	l = sqrt((WALKER_1.center.x - WALKER_2.center.x)*(WALKER_1.center.x -
WALKER_2.center.x) + (WALKER_1.center.y -
WALKER_2.center.y)*(WALKER_1.center.y - WALKER_2.center.y) +
(WALKER_1.center.z - WALKER_2.center.z)*(WALKER_1.center.z -
WALKER_2.center.z));
	return l;
}
double getDistance1D(walker WALKER_1, walker WALKER_2) {
	double l = 0.0;
	l = sqrt((WALKER_1.center.x - WALKER_2.center.x)*(WALKER_1.center.x -
WALKER_2.center.x));
	return l;
}
double getSoftDomainASurfaceDistance(walker WALKER_NEW, soft_domain_a
SPDA) {
	double l1 = 0.0;
	double l2 = 0.0;
	double l = 0.0;
	l1 = sqrt((WALKER_NEW.center.x -
SPDA.center1.x)*(WALKER_NEW.center.x - SPDA.center1.x) +
(WALKER_NEW.center.y - SPDA.center1.y)*(WALKER_NEW.center.y -
SPDA.center1.y) + (WALKER_NEW.center.z -
SPDA.center1.z)*(WALKER_NEW.center.z - SPDA.center1.z));
	l1 = l1 - SPDA.center1.r - WALKER_NEW.radius;
	l2 = sqrt((WALKER_NEW.center.x -
SPDA.center2.x)*(WALKER_NEW.center.x - SPDA.center2.x) +
(WALKER_NEW.center.y - SPDA.center2.y)*(WALKER_NEW.center.y -
SPDA.center2.y) + (WALKER_NEW.center.z -
SPDA.center2.z)*(WALKER_NEW.center.z - SPDA.center2.z));
        l2 = l2 - SPDA.center2.r - WALKER_NEW.radius;
        if (l1 - l2 < eps) {l = l1;}
        else {l = l2;}
        return l;
}
double getSoftDomainASurfaceDistance1D(walker WALKER_NEW, soft_domain_a
SPDA) {
        double l1 = 0.0;
        double l2 = 0.0;
        double l = 0.0;
        l1 = sqrt((WALKER_NEW.center.x -
SPDA.center1.x)*(WALKER_NEW.center.x - SPDA.center1.x));
        l1 = l1 - SPDA.center1.r - WALKER_NEW.radius;
        l2 = sqrt((WALKER_NEW.center.x -
SPDA.center2.x)*(WALKER_NEW.center.x - SPDA.center2.x));
        l2 = l2 - SPDA.center2.r - WALKER_NEW.radius;
        if (l1 - l2 < eps) {l = l1;}
        else {l = l2;}
        return l;
}
double getSoftDomainACentertoCenterDistance(walker WALKER_NEW,
soft_domain_a SPDA) {
        double l1 = 0.0;
        double l2 = 0.0;
        double l = 0.0;
        l1 = sqrt((WALKER_NEW.center.x -
SPDA.center1.x)*(WALKER_NEW.center.x - SPDA.center1.x) +
(WALKER_NEW.center.y - SPDA.center1.y)*(WALKER_NEW.center.y -
SPDA.center1.y) + (WALKER_NEW.center.z -
SPDA.center1.z)*(WALKER_NEW.center.z - SPDA.center1.z));
        l2 = sqrt((WALKER_NEW.center.x -
SPDA.center2.x)*(WALKER_NEW.center.x - SPDA.center2.x) +
(WALKER_NEW.center.y - SPDA.center2.y)*(WALKER_NEW.center.y -
SPDA.center2.y) + (WALKER_NEW.center.z -
SPDA.center2.z)*(WALKER_NEW.center.z - SPDA.center2.z));
        if (l1 - l2 < eps) {l = l1;}
        else {l = l2;}
        return l;
}
double getSoftDomainACentertoCenterDistance1D(walker WALKER_NEW,
soft_domain_a SPDA) {
        double l1 = 0.0;
        double l2 = 0.0;
        double l = 0.0;
        l1 = sqrt((WALKER_NEW.center.x -
SPDA.center1.x)*(WALKER_NEW.center.x - SPDA.center1.x));
        l2 = sqrt((WALKER_NEW.center.x -
SPDA.center2.x)*(WALKER_NEW.center.x - SPDA.center2.x));
        if (l1 - l2 < eps) {l = l1;}
        else {l = l2;}
        return l;
}
double getEstimateRadius(walker WALKER) {
        double radius = 0.0;
        if (WALKER.type == 0) {
                if (WALKER.component == 1) {radius = FE_BCC_A;}
                if (WALKER.component == 2) {radius = FE_BCC_A * 2.0;}
                if (WALKER.component == 3) {radius = FE_BCC_A * 2.5;}
                if (WALKER.component > 3) {radius = VACANCY_MODIFIER *
pow((WALKER.component*pow(FE_BCC_A, 3.0)), 1.0 / 3.0);}
        }
        if (WALKER.type == 1) {
                if (WALKER.component == 1) {radius = FE_RADIUS;}
                if (WALKER.component == 2) {radius = FE_RADIUS * 2.0;}
                if (WALKER.component == 3) {radius = FE_RADIUS * 2.5;}
                if (WALKER.component > 3) {radius = INTERST_MODIFIER *
pow((WALKER.component*pow(FE_RADIUS, 3.0)), 1.0 / 3.0);}
        }
        return radius;
}
double getHardDomainCenterDistance(hard_domain DOMAIN_1, hard_domain
DOMAIN_2) {
        double distance = 0.0;
        distance = sqrt((DOMAIN_1.center.x -
DOMAIN_2.center.x)*(DOMAIN_1.center.x - DOMAIN_2.center.x) +
(DOMAIN_1.center.y - DOMAIN_2.center.y)*(DOMAIN_1.center.y -
DOMAIN_2.center.y) + (DOMAIN_1.center.z -
DOMAIN_2.center.z)*(DOMAIN_1.center.z - DOMAIN_2.center.z));
        return distance;
}
long getNeighborDomainID(location DOMAIN_CENTER) {return 0;}
double getHardDomainSurfaceDistance1D(walker WALKER, hard_domain HPD) {
        double distance = 0.0;
        distance = sqrt((WALKER.center.x - HPD.center.x)*(WALKER.center.x -
HPD.center.x)) - HPD.radius - WALKER.radius;
```

```
	return distance;
}
double getHardDomainSurfaceDistance3D(walker WALKER, hard_domain HPD) {
	double distance = 0.0;
	distance = sqrt((WALKER.center.x - HPD.center.x)*(WALKER.center.x -
HPD.center.x) + (WALKER.center.y - HPD.center.y)*(WALKER.center.y -
HPD.center.y) + (WALKER.center.z - HPD.center.z)*(WALKER.center.z -
HPD.center.z)) - HPD.radius - WALKER.radius;
	return distance;
}

#endif // !TOOLBOX_H
```

File: Main.cpp

```
// Version 1 alpha demo
// J. Fan 2018-2019
#include "Main.h"
#include "Parameters.h"
#include "Toolbox.h"
using namespace std;
namespace debugFan {double temp_dv, temp_rd, temp_rw, temp_ts;}
namespace currentEvent {
        long domainID;
        int domain_type;
        double timestamp;
        int event_type;
        location afterward_location;
        int flag_s0_in_SPDa;
        double walker_distance;
        double min_distance;
        double neighbour_allow_r;
        double O1_allow_r;
        double O1_neighour_allow_r;
        long O1_neighbout_ID;
        double O2_allow_r;
        double O3_allow_r;
        long near_domain_ID;
        int near_domain_type; // 0 HPD; 1 SPDa
        double update_spda_time_temp;}
void checkWalkerOverlapping() {
        double walkerDistance;
        double walkerOccupation;
        for (i = 0;i < inputNumber; i++) {
        for (j = i + 1;j < inputNumber; j++) {
        walkerDistance = getDistance(ORI_WALKERS[i + 1], ORI_WALKERS[j + 1]);
        walkerOccupation = ORI_WALKERS[i + 1].radius + ORI_WALKERS[j +
1].radius;
        if (walkerDistance - walkerOccupation < eps) {
        ORI_WALKERS[i + 1].type = -1;
        WALKER_COUNTER = WALKER_COUNTER - 1;
        break;}}}}
void initialSoftDomainA3D(long ID, walker WALKERS_1, walker WALKERS_2) {
        double surfaceDistance;
        surfaceDistance = getDistance(WALKERS_1, WALKERS_2) -
WALKERS_1.radius - WALKERS_2.radius;
        SOFT_PD_A[ID].ID = ID;
        SOFT_PD_A[ID].center_distance = surfaceDistance;
        SOFT_PD_A[ID].clock = 0.0;
        SOFT_PD_A[ID].events_type = 10;
        SOFT_PD_A[ID].current_member = 2;
        SOFT_PD_A[ID].member_list.member[1] = WALKERS_1;
        SOFT_PD_A[ID].member_list.member[2] = WALKERS_2;
        SOFT_PD_A[ID].center1.x = WALKERS_1.center.x;
        SOFT_PD_A[ID].center1.y = WALKERS_1.center.y;
        SOFT_PD_A[ID].center1.z = WALKERS_1.center.z;
        SOFT_PD_A[ID].center1.r = WALKERS_1.radius;
        SOFT_PD_A[ID].center2.x = WALKERS_2.center.x;
        SOFT_PD_A[ID].center2.y = WALKERS_2.center.y;
        SOFT_PD_A[ID].center2.z = WALKERS_2.center.z;
        SOFT_PD_A[ID].center2.r = WALKERS_2.radius;
        ORI_WALKERS[WALKERS_1.ID].statue = 1;
        ORI_WALKERS[WALKERS_2.ID].statue = 1;
        ORI_WALKERS[WALKERS_1.ID].regionID = ID;
        ORI_WALKERS[WALKERS_2.ID].regionID = ID;
        SPDA_COUNTER = SPDA_COUNTER + 1; // Optional}
void initialUpdateSoftDomainA3D(long ID, walker WALKERS_NEW) {
        double l1 = 0.0;
        double l2 = 0.0;
        double surfaceDistance;
        double domainEdgeDistance;
        double newEdgeDistance;
        domainEdgeDistance = SOFT_PD_A[ID].center_distance +
2.0*SOFT_PD_A[ID].center1.r + 2.0*SOFT_PD_A[ID].center2.r;
        l1 = sqrt((WALKERS_NEW.center.x -
SOFT_PD_A[ID].center1.x)*(WALKERS_NEW.center.x - SOFT_PD_A[ID].center1.x)
+ (WALKERS_NEW.center.y -
SOFT_PD_A[ID].center1.y)*(WALKERS_NEW.center.y - SOFT_PD_A[ID].center1.y)
+ (WALKERS_NEW.center.z -
SOFT_PD_A[ID].center1.z)*(WALKERS_NEW.center.z -
SOFT_PD_A[ID].center1.z));
        l1 = l1 - SOFT_PD_A[ID].center1.r - WALKERS_NEW.radius;
        l2 = sqrt((WALKERS_NEW.center.x -
SOFT_PD_A[ID].center2.x)*(WALKERS_NEW.center.x - SOFT_PD_A[ID].center2.x)
+ (WALKERS_NEW.center.y -
SOFT_PD_A[ID].center2.y)*(WALKERS_NEW.center.y - SOFT_PD_A[ID].center2.y)
+ (WALKERS_NEW.center.z -
SOFT_PD_A[ID].center2.z)*(WALKERS_NEW.center.z -
SOFT_PD_A[ID].center2.z));
        l2 = l2 - SOFT_PD_A[ID].center2.r - WALKERS_NEW.radius;
        if (l1 - l2 < eps) {
```

```
        surfaceDistance = l1;
        newEdgeDistance = surfaceDistance + 2.0*WALKERS_NEW.radius +
2.0*SOFT_PD_A[ID].center1.r;
        if (newEdgeDistance - domainEdgeDistance > eps) {
        SOFT_PD_A[ID].center2.x = WALKERS_NEW.center.x;
        SOFT_PD_A[ID].center2.y = WALKERS_NEW.center.y;
        SOFT_PD_A[ID].center2.z = WALKERS_NEW.center.z;
        SOFT_PD_A[ID].center2.r = WALKERS_NEW.radius;
        SOFT_PD_A[ID].center_distance = surfaceDistance;}}
        else {
        surfaceDistance = l2;
        newEdgeDistance = surfaceDistance + 2.0*WALKERS_NEW.radius +
2.0*SOFT_PD_A[ID].center2.r;
        if (newEdgeDistance - domainEdgeDistance > eps) {
        SOFT_PD_A[ID].center1.x = WALKERS_NEW.center.x;
        SOFT_PD_A[ID].center1.y = WALKERS_NEW.center.y;
        SOFT_PD_A[ID].center1.z = WALKERS_NEW.center.z;
        SOFT_PD_A[ID].center1.r = WALKERS_NEW.radius;
        SOFT_PD_A[ID].center_distance = surfaceDistance;}}
        // Update Domain
        SOFT_PD_A[ID].current_member = SOFT_PD_A[ID].current_member + 1;
        SOFT_PD_A[ID].member_list.member[SOFT_PD_A[ID].current_member] =
WALKERS_NEW;
        ORI_WALKERS[WALKERS_NEW.ID].statue = 1;
        ORI_WALKERS[WALKERS_NEW.ID].regionID = ID;}
void checkInitialSoftDomains3D() {
        double surfaceDistance;
        for (i = 0;i < inputNumber;i++) {
        if (ORI_WALKERS[i + 1].type == -1 || ORI_WALKERS[i + 1].statue == 1) {
continue; }
        for (j = i + 1;j < inputNumber; j++) {
        if (ORI_WALKERS[j + 1].type == -1 || ORI_WALKERS[j + 1].statue == 1) {
continue; }
        surfaceDistance = getDistance(ORI_WALKERS[i + 1], ORI_WALKERS[j + 1])
- ORI_WALKERS[i + 1].radius - ORI_WALKERS[j + 1].radius;
        if (surfaceDistance - THRES_2 < eps) {
        cout << "Find one type 1 SPD during initialization." << endl;
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        initialSoftDomainA3D(SPDOMAINA_INDEX, ORI_WALKERS[i + 1],
ORI_WALKERS[j + 1]);
        for (k = j + 1;k < inputNumber; k++) {
        if (ORI_WALKERS[k + 1].type == -1 || ORI_WALKERS[k + 1].statue == 1) {
continue; }
```

```
        surfaceDistance = getSoftDomainASurfaceDistance(ORI_WALKERS[k + 1],
SOFT_PD_A[SPDOMAINA_INDEX]);
        if (surfaceDistance - THRES_2 < eps) {
        // Update Current Type 1 SPD
        initialUpdateSoftDomainA3D(SPDOMAINA_INDEX, ORI_WALKERS[k +
1]);}}}}}}
void initialSoftDomainA1D(long ID, walker WALKERS_1, walker WALKERS_2) {
        double surfaceDistance;
        surfaceDistance = getDistance1D(WALKERS_1, WALKERS_2) -
WALKERS_1.radius - WALKERS_2.radius;
        SOFT_PD_A[ID].ID = ID;
        SOFT_PD_A[ID].center_distance = surfaceDistance;
        SOFT_PD_A[ID].clock = 0.0;
        SOFT_PD_A[ID].events_type = 10;
        SOFT_PD_A[ID].current_member = 2;
        SOFT_PD_A[ID].member_list.member[1] = WALKERS_1;
        SOFT_PD_A[ID].member_list.member[2] = WALKERS_2;
        SOFT_PD_A[ID].center1.x = WALKERS_1.center.x;
        SOFT_PD_A[ID].center1.y = WALKERS_1.center.y;
        SOFT_PD_A[ID].center1.z = WALKERS_1.center.z;
        SOFT_PD_A[ID].center1.r = WALKERS_1.radius;
        SOFT_PD_A[ID].center2.x = WALKERS_2.center.x;
        SOFT_PD_A[ID].center2.y = WALKERS_2.center.y;
        SOFT_PD_A[ID].center2.z = WALKERS_2.center.z;
        SOFT_PD_A[ID].center2.r = WALKERS_2.radius;
        ORI_WALKERS[WALKERS_1.ID].statue = 1;
        ORI_WALKERS[WALKERS_2.ID].statue = 1;
        ORI_WALKERS[WALKERS_1.ID].regionID = ID;
        ORI_WALKERS[WALKERS_2.ID].regionID = ID;
        SPDA_COUNTER = SPDA_COUNTER + 1; // Optional}
void initialUpdateSoftDomainA1D(long ID, walker WALKERS_NEW) {
        double l1 = 0.0;
        double l2 = 0.0;
        double surfaceDistance;
        double domainEdgeDistance;
        double newEdgeDistance;
        // Update Center
        domainEdgeDistance = SOFT_PD_A[ID].center_distance +
2.0*SOFT_PD_A[ID].center1.r + 2.0*SOFT_PD_A[ID].center2.r;
        l1 = sqrt((WALKERS_NEW.center.x -
SOFT_PD_A[ID].center1.x)*(WALKERS_NEW.center.x -
SOFT_PD_A[ID].center1.x));
        l1 = l1 - SOFT_PD_A[ID].center1.r - WALKERS_NEW.radius;
        l2 = sqrt((WALKERS_NEW.center.x - 
SOFT_PD_A[ID].center2.x)*(WALKERS_NEW.center.x - 
SOFT_PD_A[ID].center2.x));
        l2 = l2 - SOFT_PD_A[ID].center2.r - WALKERS_NEW.radius;
        if (l1 - l2 < eps) {
        surfaceDistance = l1;
        newEdgeDistance = surfaceDistance + 2.0*WALKERS_NEW.radius + 
2.0*SOFT_PD_A[ID].center1.r;
        if (newEdgeDistance - domainEdgeDistance > eps) {
        // Change center 2 to New
        SOFT_PD_A[ID].center2.x = WALKERS_NEW.center.x;
        SOFT_PD_A[ID].center2.r = WALKERS_NEW.radius;
        SOFT_PD_A[ID].center_distance = surfaceDistance;}}
        else {
        surfaceDistance = l2;
        newEdgeDistance = surfaceDistance + 2.0*WALKERS_NEW.radius + 
2.0*SOFT_PD_A[ID].center2.r;
        if (newEdgeDistance - domainEdgeDistance > eps) {
        // Change center 1 to New
        SOFT_PD_A[ID].center1.x = WALKERS_NEW.center.x;
        SOFT_PD_A[ID].center1.r = WALKERS_NEW.radius;
        SOFT_PD_A[ID].center_distance = surfaceDistance;}}
        // Update Domain
        SOFT_PD_A[ID].current_member = SOFT_PD_A[ID].current_member + 1;
        SOFT_PD_A[ID].member_list.member[SOFT_PD_A[ID].current_member] = 
WALKERS_NEW;
        ORI_WALKERS[WALKERS_NEW.ID].statue = 1;
        ORI_WALKERS[WALKERS_NEW.ID].regionID = ID;}
void checkInitialSoftDomains1D() {
        double surfaceDistance;
        for (i = 0;i < inputNumber;i++) {
        if (ORI_WALKERS[i + 1].type != -1 && ORI_WALKERS[i + 1].statue != 1) {
        for (j = i + 1;j < inputNumber; j++) {
        if (ORI_WALKERS[j + 1].type != -1 && ORI_WALKERS[j + 1].statue != 1){
        surfaceDistance = getDistance1D(ORI_WALKERS[i + 1], ORI_WALKERS[j + 
1]) - ORI_WALKERS[i + 1].radius - ORI_WALKERS[j + 1].radius;
        if (surfaceDistance - THRES_2 < eps) {
        cout << "Find one type 1 SPD during initialization." << endl;
        // Create New Type 1 SPD
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        initialSoftDomainA1D(SPDOMAINA_INDEX, ORI_WALKERS[i + 1], 
ORI_WALKERS[j + 1]);
         for (k = j + 1;k < inputNumber; k++) {
        if (ORI_WALKERS[k + 1].type != -1 && ORI_WALKERS[k + 1].statue != 1){
```

```
        surfaceDistance = getSoftDomainASurfaceDistance1D(ORI_WALKERS[k + 1],
SOFT_PD_A[SPDOMAINA_INDEX]);
        if (surfaceDistance - THRES_2 < eps) {
        initialUpdateSoftDomainA1D(SPDOMAINA_INDEX, ORI_WALKERS[k +
1]);}}}}}}}}}
void initialHardDomain3D(long ID, walker WALKER, double DOMAIN_R) {
        HARD_PD[ID].ID = ID;
        HARD_PD[ID].center.x = WALKER.center.x;
        HARD_PD[ID].center.y = WALKER.center.y;
        HARD_PD[ID].center.z = WALKER.center.z;
        HARD_PD[ID].clock = 0.0;
        HARD_PD[ID].events_type = 1;
        HARD_PD[ID].radius = DOMAIN_R;
        HARD_PD[ID].walker_ID = WALKER.ID;
        HARD_PD[ID].walker_type = WALKER.type;
        ORI_WALKERS[WALKER.ID].statue = 2;
        ORI_WALKERS[WALKER.ID].regionID = ID;
        HPD_COUNTER = HPD_COUNTER + 1;}
void setInitialHardDomains3D() {
        double domainRadius;
        for (i = 0;i < inputNumber;i++) {
        if (ORI_WALKERS[i + 1].type != -1 && ORI_WALKERS[i + 1].statue == 0) {
        domainRadius = sqrt(INIDOMAIN_MODIFIER*ORI_WALKERS[i +
1].diffusivity*INITIAL_EVENTS_TIME) + ORI_WALKERS[i + 1].radius;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        initialHardDomain3D(HPDOMAIN_INDEX, ORI_WALKERS[i + 1],
domainRadius);}}}
void initialHardDomain1D(long ID, walker WALKER, double DOMAIN_RS) {
        HARD_PD[ID].ID = ID;
        HARD_PD[ID].center.x = WALKER.center.x;
        HARD_PD[ID].center.y = 0.0;
        HARD_PD[ID].center.z = 0.0;
        HARD_PD[ID].clock = 0.0;
        HARD_PD[ID].events_type = 1;
        HARD_PD[ID].radius = DOMAIN_RS;
        HARD_PD[ID].walker_ID = WALKER.ID;
        HARD_PD[ID].walker_type = WALKER.type;
        ORI_WALKERS[WALKER.ID].statue = 2;
        ORI_WALKERS[WALKER.ID].regionID = ID;
        HPD_COUNTER = HPD_COUNTER + 1;}
void setInitialHardDomains1D() {
        double domainRadius;
        for (i = 0;i < inputNumber;i++) {
        if (ORI_WALKERS[i + 1].type != -1 && ORI_WALKERS[i + 1].statue == 0) {
        domainRadius = sqrt(INIDOMAIN_MODIFIER*ORI_WALKERS[i +
1].diffusivity*INITIAL_EVENTS_TIME) + ORI_WALKERS[i + 1].radius;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        initialHardDomain1D(HPDOMAIN_INDEX,ORI_WALKERS[i+1],domainRadi
us);}}}
double getHardDomainModifier(hard_domain DOMAIN_1, hard_domain
DOMAIN_2){
        double l, ld;
        double fraction = 1.0;
        l = getHardDomainCenterDistance(DOMAIN_1, DOMAIN_2);
        ld = DOMAIN_1.radius + DOMAIN_2.radius;
        if (ld - l > eps) {fraction = l / ld;}
        return fraction;}
void checkInitialDomain1D() {
        double centerDistance, centerDistance1, centerDistance2;
        double minDistance = 999999999.0;
        double fix;
        int minIndex;
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        for (j = i + 1;j < HPDOMAIN_INDEX;j++) {
        fix = getHardDomainModifier(HARD_PD[i + 1], HARD_PD[j + 1]);
        HARD_PD[i + 1].radius = fix*HARD_PD[i + 1].radius;
        HARD_PD[j + 1].radius = fix*HARD_PD[j + 1].radius;
        if (HARD_PD[i + 1].radius - ORI_WALKERS[HARD_PD[i +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[i + 1].walker_ID << ", HPD ID " <<
HARD_PD[i + 1].ID << endl;
        system("pause");
        exit(0);}
        if (HARD_PD[j + 1].radius - ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[j + 1].walker_ID << ", HPD ID " <<
HARD_PD[j + 1].ID << endl;
        system("pause");
        exit(0);}}
        // Find the most close SPD center, search all SPDa
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        centerDistance1 = sqrt((HARD_PD[i + 1].center.x - SOFT_PD_A[j +
1].center1.x)*(HARD_PD[i + 1].center.x - SOFT_PD_A[j + 1].center1.x));
        centerDistance2 = sqrt((HARD_PD[i + 1].center.x - SOFT_PD_A[j +
1].center2.x)*(HARD_PD[i + 1].center.x - SOFT_PD_A[j + 1].center2.x));
```

```
        if (centerDistance1 - centerDistance2 < eps) { centerDistance = centerDistance1;
}
        else { centerDistance = centerDistance2; }
        if (centerDistance - minDistance < eps) {
        minDistance = centerDistance;
        minIndex = j + 1;}}
        if (minDistance - HARD_PD[i + 1].radius < eps) {
        centerDistance1 = sqrt((HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center1.x)*(HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center1.x));
        centerDistance2 = sqrt((HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center2.x)*(HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center2.x));
        if (centerDistance1 - centerDistance2 < eps) { HARD_PD[i + 1].radius =
centerDistance1;}
        else { HARD_PD[i + 1].radius = centerDistance2; }
        if (HARD_PD[i + 1].radius - ORI_WALKERS[HARD_PD[i +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[i + 1].walker_ID << ", HPD ID " <<
HARD_PD[i + 1].ID << endl;
        system("pause");
        exit(0);}}}}
void checkInitialDomain3D() {
        double centerDistance, centerDistance1, centerDistance2;
        double minDistance = 999999999.0;
        double fix;
        int minIndex;
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        // For all the rest HPD, modify their domains, after this HPD_i will not be
overlapped with any HPD
        for (j = i + 1;j < HPDOMAIN_INDEX;j++) {
        fix = getHardDomainModifier(HARD_PD[i + 1], HARD_PD[j + 1]);
        HARD_PD[i + 1].radius = fix*HARD_PD[i + 1].radius;
        HARD_PD[j + 1].radius = fix*HARD_PD[j + 1].radius;
        if (HARD_PD[i + 1].radius - ORI_WALKERS[HARD_PD[i +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[i + 1].walker_ID << ", HPD ID " <<
HARD_PD[i + 1].ID << endl;
        system("pause");
        exit(0);}
        if (HARD_PD[j + 1].radius - ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[j + 1].walker_ID << ", HPD ID " <<
HARD_PD[j + 1].ID << endl;
        system("pause");
        exit(0);}}
        // Find the most close SPD center, search all SPDa
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        centerDistance1 = sqrt((HARD_PD[i + 1].center.x - SOFT_PD_A[j +
1].center1.x)*(HARD_PD[i + 1].center.x - SOFT_PD_A[j + 1].center1.x)+
(HARD_PD[i + 1].center.y - SOFT_PD_A[j + 1].center1.y)*(HARD_PD[i + 1].center.y
- SOFT_PD_A[j + 1].center1.y)+ (HARD_PD[i + 1].center.z - SOFT_PD_A[j +
1].center1.z)*(HARD_PD[i + 1].center.z - SOFT_PD_A[j + 1].center1.z));
        centerDistance2 = sqrt((HARD_PD[i + 1].center.x - SOFT_PD_A[j +
1].center2.x)*(HARD_PD[i + 1].center.x - SOFT_PD_A[j + 1].center2.x)+
(HARD_PD[i + 1].center.y - SOFT_PD_A[j + 1].center2.y)*(HARD_PD[i + 1].center.y
- SOFT_PD_A[j + 1].center2.y)+ (HARD_PD[i + 1].center.z - SOFT_PD_A[j +
1].center2.z)*(HARD_PD[i + 1].center.z - SOFT_PD_A[j + 1].center2.z));
        if (centerDistance1 - centerDistance2 < eps) { centerDistance = centerDistance1;
}
        else { centerDistance = centerDistance2; }
        if (centerDistance - minDistance < eps) {
        minDistance = centerDistance;
        minIndex = j + 1;}}
        // Adjust HPD
        if (minDistance - HARD_PD[i + 1].radius < eps) {
        centerDistance1 = sqrt((HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center1.x)*(HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center1.x)+ (HARD_PD[i + 1].center.y -
SOFT_PD_A[minIndex].center1.y)*(HARD_PD[i + 1].center.y -
SOFT_PD_A[minIndex].center1.y)+ (HARD_PD[i + 1].center.z -
SOFT_PD_A[minIndex].center1.z)*(HARD_PD[i + 1].center.z -
SOFT_PD_A[minIndex].center1.z));
        centerDistance2 = sqrt((HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center2.x)*(HARD_PD[i + 1].center.x -
SOFT_PD_A[minIndex].center2.x)+ (HARD_PD[i + 1].center.y -
SOFT_PD_A[minIndex].center2.y)*(HARD_PD[i + 1].center.y -
SOFT_PD_A[minIndex].center2.y)+ (HARD_PD[i + 1].center.z -
SOFT_PD_A[minIndex].center2.z)*(HARD_PD[i + 1].center.z -
SOFT_PD_A[minIndex].center2.z));
        if (centerDistance1 - centerDistance2 < eps) { HARD_PD[i + 1].radius =
centerDistance1; }
        else { HARD_PD[i + 1].radius = centerDistance2; }
        if (HARD_PD[i + 1].radius - ORI_WALKERS[HARD_PD[i +
1].walker_ID].radius < eps) {
        cout << "Fatal Error 900: Cannot maintain a valid HPD. Exit." << endl;
        cout << "Walker ID " << HARD_PD[i + 1].walker_ID << ", HPD ID " <<
HARD_PD[i + 1].ID << endl;
        system("pause");
        exit(0);}}}}
int recheckInitialDomain() {
        int checkFlag;
        checkFlag = 0;
        return checkFlag;}
void buildHardDomain(long ID, walker WALKERS) {
        cout << "Refer to ASMS" << endl;}
void buildSoftDomainA(long ID, walker WALKERS_1, walker WALKERS_2) {
        cout << "Refer to ASMS" << endl;}
void updateSoftDomainA(long ID, walker NEW_WALKER) {
        cout << "Refer to ASMS" << endl;}

int main(){
        // Welcome to PROJECT aCRD
        cout << "//////////////////////////////////////////////////////////////" << endl;
        cout << "The following Caluculation is Based on Version " <<
MAJOR_VERSION_NUMBER << "." << MINOR_VERSION_NUMBER << "." <<
REVISIONB_NUMBER << " [" << VERSION_ID << "]" << endl;
        cout << "PPOJECT advanced Cluster Reaction Dynamics" << endl;
        cout << "PROJECT aCRD" << endl;
        cout << "//////////////////////////////////////////////////////////////" << endl;
        // Initilization
        cout << "System Initilization" << endl;
        // Module Loading and Integrity Check
        HMODULE dllLibDiffusivity = LoadLibrary("DiffusivityLib.dll");
        HMODULE dllSolverMajorDiff = LoadLibrary("MajorDiffSolver.dll");
        HMODULE dllSolverTransientDiff = LoadLibrary("TransientDiffSolver.dll");
        HMODULE dllSolverLocal = LoadLibrary("LocalSolver.dll");
        HMODULE dllSolverDissociation = LoadLibrary("DissociationSolver.dll");
        if (dllLibDiffusivity == NULL) {
        cerr << "Cannot find DiffusivityLib.dll. Please check framework integrity." <<
endl;
        releaseModule();
        return -1;}
        if (dllSolverMajorDiff == NULL) {
        cout << "Cannot find MajorDiffSolver.dll. Please check framework integrity."
<< endl;
        releaseModule();
        return -1;}
        if (dllSolverTransientDiff == NULL) {
```

```
        cout << "Cannot find TransientDiffSolver.dll. Please check framework integrity."
<< endl;
        releaseModule();
        return -1;}
        if (dllSolverLocal == NULL) {
        cout << "Cannot find LocalSolver.dll. Please check framework integrity." <<
endl;
        releaseModule();
        return -1;}
        if (dllSolverDissociation == NULL) {
        cout << "Cannot find DissociationSolver.dll. Please check framework integrity."
<< endl;
        releaseModule();
        return -1;}
        cout << "Finish Loading Framework." << endl;
        cout << endl;
        // Function Initilization
        typedef double(*nModuleVersion);
        // Loading Library
        cout << "Begin to Load Library Module." << endl;
        // Loading Standard Diffusivity Library
        // Rely on Diffusivity Lib (DiffusivityLib.dll)
        // Contain 1 Function: getMajorDiffTimeStamp and getDvValue
        nModuleVersion getDiffusivityLibVer =
(nModuleVersion)GetProcAddress(dllLibDiffusivity, TEXT("nDvLibVersion"));
        cout << "Framework Load Version " << *getDiffusivityLibVer << " Diffusivity
Library." << endl;
        typedef double(*functionLibDv)(double, int, long);
        // Function double getDvValue(double E, int type, long size)
        // Diffusivity: nm^2/s
        // Energy unit in keV, Type [int] (0 for V, 1 for I), Size [Long] (Cluster)
        functionLibDv getDvValue = (functionLibDv)GetProcAddress(dllLibDiffusivity,
TEXT("getDvValue"));
        if (getDvValue == NULL) {
        cout << "Cannot link to the function getDvValue. Please check the version of
DiffusivityLib.dll." << endl;
        releaseModule();
        return -1;}
        cout << "Loading Diffusivity Library... Complete." << endl;
        // Full Lib for all parameter (radius, crystal, etc): TODO
        cout << "Finish Loading Library Module." << endl;
        cout << endl;
        // Loading Solver
        cout << "Begin to Load Solver Module." << endl;
        // Loading Event 01 Solver from dllSolverMajorDiff
        // Rely on Major Diffusion Solver (MajorDiffSolver.dll)
        // Contain 4 Functions: getMajorDiffTimeStamp1D, getMajorDiffTimeStamp3D,
getMajorDiffRelativeLocation1D and getMajorDiffRelativeLocation3D
        nModuleVersion getE01SolverVer =
(nModuleVersion)GetProcAddress(dllSolverMajorDiff, TEXT("nE01SolverVersion"));
        cout << "Framework Load Version " << *getE01SolverVer << " Major Diffusion
Solver." << endl;
        typedef double(*functionE01_1)(double, double, double);
        typedef struct location(*functionE01_2)(double, double, double, double,
location);
        // Function double getMajorDiffTimeStamp1D(double Dv, double
domainRadius, double walkerRadius)
        // TimeStamp: s
        // Dv unit in nm^2/s, Radius unit in nm
        functionE01_1 getMajorDiffTimeStamp1D =
(functionE01_1)GetProcAddress(dllSolverMajorDiff,
TEXT("getMajorDiffTimeStamp1D"));
        // Function double getMajorDiffTimeStamp3D(double Dv, double
domainRadius, double walkerRadius)
        // TimeStamp: s
        // Dv unit in nm^2/s, Radius unit in nm
        functionE01_1 getMajorDiffTimeStamp3D =
(functionE01_1)GetProcAddress(dllSolverMajorDiff,
TEXT("getMajorDiffTimeStamp3D"));
        // Function location getMajorDiffRelativeLocation1D(double Dv, double
timestamp, double domainRadius, double walkerRadius, location centerLocation)
        // Location unit in nm
        // Dv unit in nm^2/s, time unit in s, Radius unit in nm, location unit in nm
        functionE01_2 getMajorDiffRelativeLocation1D =
(functionE01_2)GetProcAddress(dllSolverMajorDiff,
TEXT("getMajorDiffRelativeLocation1D"));
        // Function location getMajorDiffRelativeLocation3D(double Dv, double
timestamp, double domainRadius, double walkerRadius, location centerLocation)
        // Location unit in nm
        // Dv unit in nm^2/s, times unit is s, Radius unit in nm, location unit in nm
        functionE01_2 getMajorDiffRelativeLocation3D =
(functionE01_2)GetProcAddress(dllSolverMajorDiff,
TEXT("getMajorDiffRelativeLocation3D"));
        if (getMajorDiffTimeStamp1D == NULL) {
        cout << "Cannot link to the function getMajorDiffTimeStamp1D. Please check
the version of MajorDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getMajorDiffTimeStamp3D == NULL) {
        cout << "Cannot link to the function getMajorDiffTimeStamp3D. Please check
the version of MajorDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getMajorDiffRelativeLocation1D == NULL) {
        cout << "Cannot link to the function getMajorDiffRelativeLocation1D. Please
check the version of MajorDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getMajorDiffRelativeLocation3D == NULL) {
        cout << "Cannot link to the function getMajorDiffRelativeLocation3D. Please
check the version of MajorDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        cout << "Loading Major Diffusion (E01) Solver Module... Complete." << endl;
        // Loading Event 02 Solver from dllSolverTransientDiff
        // Rely on Transient Diffusion Solver (TransientDiffSolver.dll)
        // Contain 2 Functions: getTransDiffRelativeLocation1D and
getTransDiffRelativeLocation3D
        nModuleVersion getE02SolverVer =
(nModuleVersion)GetProcAddress(dllSolverTransientDiff,
TEXT("nE02SolverVersion"));
        cout << "Framework Load Version " << *getE02SolverVer << " Transient
Diffusion Solver." << endl;
        typedef struct location(*functionE02_1)(double, double, double, double,
location);
        // Function location getTransDiffRelativeLocation1D(double Dv, double
timestamp, double domainRadius, double walkerRadius, location centerLocation)
        // Location unit in nm
        // Time unit in s
        // Dv unit in nm^2/s, Radius unit in nm, location unit in nm
        functionE02_1 getTransDiffRelativeLocation1D =
(functionE02_1)GetProcAddress(dllSolverTransientDiff,
TEXT("getTransDiffRelativeLocation1D"));
        // Function location getTransDiffRelativeLocation3D(double Dv, double
timestamp, double domainRadius, double walkerRadius, location centerLocation)
        // Location unit in nm
        // Time unit in s
        // Dv unit in nm^2/s, Radius unit in nm, location unit in nm
        functionE02_1 getTransDiffRelativeLocation3D =
(functionE02_1)GetProcAddress(dllSolverTransientDiff,
TEXT("getTransDiffRelativeLocation3D"));
        if (getTransDiffRelativeLocation1D == NULL) {
        cout << "Cannot link to the function getTransDiffRelativeLocation1D. Please
check the version of TransientDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getTransDiffRelativeLocation3D == NULL) {
        cout << "Cannot link to the function getTransDiffRelativeLocation3D. Please
check the version of TransientDiffSolver.dll." << endl;
        releaseModule();
        return -1;}
        cout << "Loading Transient Diffusion (E02) Solver Module... Complete." <<
endl;
        // Loading Event 10 Solver from dllSolverLocal
        // Rely on Local Reaction Solver (LocalSolver.dll)
        // Annihilation only version in 0.5 while coalescence version in 1.0
        // Contain 4 Functions: getReactionTimeStamp1D, getReactionTimeStamp3D,
getReactionFinalStatues1D and getReactionFinalStatues3D
        nModuleVersion getE10SolverVer =
(nModuleVersion)GetProcAddress(dllSolverLocal, TEXT("nE10SolverVersion"));
        cout << "Framework Load Version " << *getE10SolverVer << " Local Reaction
Solver." << endl;
        typedef double(*functionE10_1)(reaction_current_walker, double);
        typedef struct reaction_final_walker(*functionE10_2)(reaction_current_walker,
int, domaincenter, domaincenter);
        // Function double getReactionTimeStamp1D(reaction_current_walker walkers,
double initial_time)
        // Input with current walker list, storaged in SPDa
        // Return with SPDa break time: s
        functionE10_1 getReactionTimeStamp1D =
(functionE10_1)GetProcAddress(dllSolverLocal, TEXT("getReactionTimeStamp1D"));
        // Function double getReactionTimeStamp3D(reaction_current_walker walkers,
double initial_time)
        // Input with current walker list and overall number, storaged in SPDa
        // Return with SPDa break time: s
        functionE10_1 getReactionTimeStamp3D =
(functionE10_1)GetProcAddress(dllSolverLocal, TEXT("getReactionTimeStamp3D"));
        // Function reaction_final_walker
getReactionFinalStatues1D(reaction_current_walker walkers, int init_member,
domaincenter centerA, domaincenter centerB)
        // Input with current walker list, overall number and center info, storaged in
SPDa
        // Return with final walker list, storage to SPDa
        functionE10_2 getReactionFinalStatues1D =
(functionE10_2)GetProcAddress(dllSolverLocal, TEXT("getReactionFinalStatues1D"));
```

```
        // Function reaction_final_walker
getReactionFinalStatues3D(reaction_current_walker walkers, int init_member,
domaincenter centerA, domaincenter centerB)
        // Input with current walker list, overall number and center info, storaged in
SPDa
        // Return with final walker list, storage to SPDa
        functionE10_2 getReactionFinalStatues3D =
(functionE10_2)GetProcAddress(dllSolverLocal, TEXT("getReactionFinalStatues3D"));
        if (getReactionTimeStamp1D == NULL) {
        cout << "Cannot link to the function getReactionTimeStamp1D. Please check the
version of LocalSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getReactionTimeStamp3D == NULL) {
        cout << "Cannot link to the function getReactionTimeStamp3D. Please check the
version of LocalSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getReactionFinalStatues1D == NULL) {
        cout << "Cannot link to the function getReactionFinalStatues1D. Please check
the version of LocalSolver.dll." << endl;
        releaseModule();
        return -1;}
        if (getReactionFinalStatues3D == NULL) {
        cout << "Cannot link to the function getReactionFinalStatues3D. Please check
the version of LocalSolver.dll." << endl;
        releaseModule();
        return -1;}
        cout << "Loading Local Reaction (E10) Solver Module... Complete." << endl;
        // Loading Event 11 Solver from dllSolverDissociation
        // Rely on Dissociation Solver (DissociationSolver.dll)
        // Contain 1 function: getDissociationFlag
        // Future contain 2 more functions: getDissociationFinalStatues1D and
getDissociationFinalStatues3D
        // May contain other 2 functions: getDissociationTimestamp1D and
getDissociationTimestamp3D
        nModuleVersion getE11SolverVer =
(nModuleVersion)GetProcAddress(dllSolverDissociation,
TEXT("nE11SolverVersion"));
        cout << "Framework Load Version " << *getE11SolverVer << " Dissociation
Solver." << endl;
        typedef int(*functionE11_1)(walker);
        // Function int getDissociationFlag(walker WALKER)
        // Input with walker
```

```
    // Return with dissociation flag, 1: E11, 0: Nothing
    functionE11_1 getDissociationFlag =
(functionE11_1)GetProcAddress(dllSolverDissociation, TEXT("getDissociationFlag"));
    if (getDissociationFlag == NULL) {
    cout << "Cannot link to the function getDissociationFlag. Please check the
version of DissociationSolver.dll." << endl;
    releaseModule();
    return -1;}
    cout << "Loading Dissociation (E11) Solver Module... Complete." << endl;
    // E20+ & Other Events
    // Currently no solver in need, all embedded with ASMS logic
    cout << "Finish Loading Solver Module." << endl;
    cout << "/////////////////////////////////////////////////////////////////////////" << endl;
    // Loading User Data
Test_Start_1:
    cout << "Begin to load User Data." << endl;
    // System Parameter
    filenameInput = "System.ini";
    fstream fileInput1(filenameInput, ios::in);
    if (!fileInput1) {
    cerr << "Cannot Open System.ini" << endl;
    system("pause");
    return -1;}
    fileInput1 >> DISPLAYFLAG_SYS >> ININOTES;
    fileInput1 >> DISPLAYFLAG_DEB >> ININOTES;
    fileInput1 >> END_CLOCK >> ININOTES;
    fileInput1 >> DUMPFLAG_INN >> ININOTES;
    fileInput1 >> DUMPLAMMPSFLAG_INN >> ININOTES;
    fileInput1 >> DUMPCOUNTER_FLAG >> ININOTES;
    fileInput1.close();
    // Documents Input Initial
    // First Line to be the totol number of input
    // Second Line to be the dimesion number
    // Third line to be the boundary (Third to Fifth line for 3D)
    // Each line (3D): Type, Cluster Number, X, Y, Z
    // ofstream fileOutput("Output.txt")
    cout << "Please Input the Filename. Located in the Working Direction. Location
Unit in nm." << endl;
    cin >> filenameInput;
    fstream fileInput(filenameInput, ios::in);
    if (!fileInput){
    cerr << "Cannot Open the File." << endl;
    system("pause");
    return -1;}
```

```
          fileInput >> inputNumber;
          fileInput >> space_dimension;
          switch (space_dimension){
          case 1:
          fileInput >> X_LOWWER_LIMIT >> X_UPPER_LIMIT;
          if (X_LOWWER_LIMIT - X_UPPER_LIMIT > eps) {
          cerr << "Error in boundary setting. Please check your input file." << endl;
          system("pause");
          return(-1);}
          for (i = 0;i < inputNumber; ++i) {
          fileInput >> ORI_WALKERS[i + 1].type >> ORI_WALKERS[i + 1].component
>> ORI_WALKERS[i + 1].center.x;
          if (ORI_WALKERS[i + 1].center.x - X_LOWWER_LIMIT<eps ||
ORI_WALKERS[i + 1].center.x - X_UPPER_LIMIT>eps) {
          cerr << "Error in data. Please check your input file." << endl;
          system("pause");
          return(-1);}
          ORI_WALKERS[i + 1].radius = getEstimateRadius(ORI_WALKERS[i + 1]);
          ORI_WALKERS[i + 1].ID = i + 1;
          ORI_WALKERS[i + 1].statue = 0;
          ORI_WALKERS[i + 1].energy = 0.0; // Not in use for current version
          ORI_WALKERS[i + 1].diffusivity = getDvValue(ORI_WALKERS[i +
1].energy, ORI_WALKERS[i + 1].type, ORI_WALKERS[i + 1].component);
          ORI_WALKERS[i + 1].timestamp = 0.0;}
          break;
          case 3:
          fileInput >> X_LOWWER_LIMIT >> X_UPPER_LIMIT;
          if (X_LOWWER_LIMIT - X_UPPER_LIMIT > eps) {
          cerr << "Error in boundary setting. Please check your input file." << endl;
          system("pause");
          return(-1);}
          fileInput >> Y_LOWWER_LIMIT >> Y_UPPER_LIMIT;
          if (Y_LOWWER_LIMIT - Y_UPPER_LIMIT > eps) {
          cerr << "Error in boundary setting. Please check your input file." << endl;
          system("pause");
          return(-1);}
          fileInput >> Z_LOWWER_LIMIT >> Z_UPPER_LIMIT;
          if (Z_LOWWER_LIMIT - Z_UPPER_LIMIT > eps) {
          cerr << "Error in boundary setting. Please check your input file." << endl;
          system("pause");
          return(-1);}
          for (i = 0;i < inputNumber; ++i) {
```

```
        fileInput >> ORI_WALKERS[i + 1].type >> ORI_WALKERS[i + 1].component
>> ORI_WALKERS[i + 1].center.x >> ORI_WALKERS[i + 1].center.y >>
ORI_WALKERS[i + 1].center.z;
        if (ORI_WALKERS[i + 1].center.x - X_LOWWER_LIMIT<eps ||
ORI_WALKERS[i + 1].center.x - X_UPPER_LIMIT>eps) {
        cerr << "Error in data. Please check your input file." << endl;
        system("pause");
        return(-1);}
        if (ORI_WALKERS[i + 1].center.y - Y_LOWWER_LIMIT<eps ||
ORI_WALKERS[i + 1].center.y - Y_UPPER_LIMIT>eps) {
        cerr << "Error in data. Please check your input file." << endl;
        system("pause");
        return(-1);}
        if (ORI_WALKERS[i + 1].center.z - Z_LOWWER_LIMIT<eps ||
ORI_WALKERS[i + 1].center.z - Z_UPPER_LIMIT>eps) {
        cerr << "Error in data. Please check your input file." << endl;
        system("pause");
        return(-1);}
        ORI_WALKERS[i + 1].radius = getEstimateRadius(ORI_WALKERS[i + 1]);
        ORI_WALKERS[i + 1].ID = i + 1;
        ORI_WALKERS[i + 1].statue = 0;
        ORI_WALKERS[i + 1].energy = 0.0; // Not in use for current version
        ORI_WALKERS[i + 1].diffusivity = getDvValue(ORI_WALKERS[i +
1].energy, ORI_WALKERS[i + 1].type, ORI_WALKERS[i + 1].component);
        ORI_WALKERS[i + 1].timestamp = 0.0;}
        break;
        default:
        cerr << "Error in dimension setting. Please check your input file." << endl;
        system("pause");
        return(-1);
        break;}
        fileInput.close();
        WALKER_COUNTER = inputNumber;
        HPDOMAIN_INDEX = 0;
        SPDOMAINA_INDEX = 0;
        SPDOMAINB_INDEX = 0;
        cout << "Total " << WALKER_COUNTER << " walkers are read from current
initial data." << endl;
        // Finish User Data. ORI_WALKERS[], WALKER_INDEX, X/Y/Z_LIMITS,
space_dimension are useful data.
        double dump_min;
        double dump_max;
        dump_min = X_LOWWER_LIMIT;
        dump_max = X_UPPER_LIMIT;
```

```
    cout << "Finish Loading User Data." << endl;
    cout << "//////////////////////////////////////////////////////////////////" << endl;
    WALKER_INDEX = inputNumber; // Reserve for New Walkers, Storage to
ORI_WALKERS, inputNumber is not supposed to be changed.
    // Initialization
    // Now all walkers are released
    // Build Initial Domains: delete walker overlap, allow HPD (E01+E11) and
SPD_a (E10). Able to fit with restart or sync.
    checkWalkerOverlapping();
    // Build SPD_a, then build HPD for the rest, run overlap check for HPD
    // In current Version, SPD_a will be set to fixed break time and annihilation only
    // The DOMAIN_INDEX is also updated, point to the new blank domain in array
    if (space_dimension == 1) {
    cout << "Begin to initial 1D space." << endl;
    checkInitialSoftDomains1D();
    setInitialHardDomains1D();
    int Flag_Initial_HPD = 1;
    while (Flag_Initial_HPD) {
    checkInitialDomain1D();
    Flag_Initial_HPD = recheckInitialDomain(); }}
    if (space_dimension == 3) {
    cout << "Begin to initial 3D space." << endl;
    checkInitialSoftDomains3D();
    setInitialHardDomains3D();
    int Flag_Initial_HPD = 1;
    while (Flag_Initial_HPD) {
    checkInitialDomain3D();
    Flag_Initial_HPD = recheckInitialDomain(); }}
    cout << endl;
    cout << "Current PROJECT aCRD main stage contains " <<
WALKER_COUNTER << " walkers in " << HPD_COUNTER << " HPD, " <<
SPDA_COUNTER << " Type 1 SPD and " << SPDB_COUNTER << " Type 2 SPD."
<< endl;
    cout << "Finish Space Initialization." << endl;
    cout << "//////////////////////////////////////////////////////////////////" << endl;
    // Introduce Solvers and Build Event Queue
Test_Start_2:
    cout << "Begin to apply solvers and set event queue." << endl;
    WORLD_CLOCK = 0.0;
    // HPD timestamp initialization and event set
    // E01 or E11 (Not in this version)
    if (space_dimension == 1) {
    for (i = 0;i < HPDOMAIN_INDEX;i++) {
    EVENT_INDEX = EVENT_INDEX + 1;
```

```
        HARD_PD[i + 1].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[i + 1].walker_ID].diffusivity,
HARD_PD[i + 1].radius, ORI_WALKERS[HARD_PD[i + 1].walker_ID].radius);
        EVENT_LIST[EVENT_INDEX].domainID = HARD_PD[i + 1].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = HARD_PD[i + 1].events_time +
WORLD_CLOCK;
        EVENT_LIST[EVENT_INDEX].eventType = 1;
        HARD_PD[i + 1].event_card = EVENT_INDEX;}}
        if (space_dimension == 3) {
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        EVENT_INDEX = EVENT_INDEX + 1;
        HARD_PD[i + 1].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[i + 1].walker_ID].diffusivity,
HARD_PD[i + 1].radius, ORI_WALKERS[HARD_PD[i + 1].walker_ID].radius);
        EVENT_LIST[EVENT_INDEX].domainID = HARD_PD[i + 1].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = HARD_PD[i + 1].events_time +
WORLD_CLOCK;
        EVENT_LIST[EVENT_INDEX].eventType = 1;
        HARD_PD[i + 1].event_card = EVENT_INDEX;}}
        // SPDa timestamp initialization and event set
        // E10
        if (space_dimension == 1) {
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        EVENT_INDEX = EVENT_INDEX + 1;
        // SPDa will trigger within INITIAL_EVENTS_TIME +
INITIAL_EVENTS_TIME * Random[0,1)
        // RANDOM_SEED = rand() / (double)(RAND_MAX);
        // SOFT_PD_A[i + 1].events_time = INITIAL_EVENTS_TIME +
INITIAL_EVENTS_TIME*RANDOM_SEED; // Replaced with solver interface once
LocalSolver is provided
        SOFT_PD_A[i + 1].events_time = getReactionTimeStamp1D(SOFT_PD_A[i +
1].member_list, INITIAL_EVENTS_TIME); // With solver interface
        EVENT_LIST[EVENT_INDEX].domainID = SOFT_PD_A[i + 1].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 1;
        EVENT_LIST[EVENT_INDEX].timestamp = SOFT_PD_A[i + 1].events_time
+ WORLD_CLOCK;
        EVENT_LIST[EVENT_INDEX].eventType = 10;
        SOFT_PD_A[i + 1].event_card = EVENT_INDEX; }}
        if (space_dimension == 3) {
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        EVENT_INDEX = EVENT_INDEX + 1;
        // This version will not call local solver, SPDa will trigger within
INITIAL_EVENTS_TIME + INITIAL_EVENTS_TIME * Random[0,1)
        // RANDOM_SEED = rand() / (double)(RAND_MAX);
        // SOFT_PD_A[i + 1].events_time = INITIAL_EVENTS_TIME +
INITIAL_EVENTS_TIME*RANDOM_SEED; // Replaced with solver interface once
LocalSolver is provided
        SOFT_PD_A[i + 1].events_time = getReactionTimeStamp3D(SOFT_PD_A[i +
1].member_list, INITIAL_EVENTS_TIME); // With solver interface
        EVENT_LIST[EVENT_INDEX].domainID = SOFT_PD_A[i + 1].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 1;
        EVENT_LIST[EVENT_INDEX].timestamp = SOFT_PD_A[i + 1].events_time
+ WORLD_CLOCK;
        EVENT_LIST[EVENT_INDEX].eventType = 10;
        SOFT_PD_A[i + 1].event_card = EVENT_INDEX; }}
        // SPDb is hybrid-driven, use world clock control or fix step events
        // Not in this version
        // E20 New Walker Event
        // Only Event Information would be applied with type=20, No need for domain
        // Use extra file to control the input infomation
        // New walker will be introduced in event queue followed by ASMS
        cout << "Begin to Load Extra Walker (irradiation) information." << endl;
        filenameInput = "Irradiation.ini";
        fstream fileInput2(filenameInput, ios::in);
        if (!fileInput2) {
        cerr << "Cannot Open Irradiation.ini" << endl;
        system("pause");
        return -1;}
        fileInput2 >> irradiationCheck >> ININOTES;
        if (irradiationCheck == 0) {
        cout << "No Extra Walker (irradiation) introduced in current calculation." <<
endl;}
        else {
        cout << "Loading Extra Walker (irradiation) information." << endl;
        fileInput2 >> irradiationPulse >> ININOTES;
        fileInput2 >> irradiationINumber >> ININOTES;
        fileInput2 >> irradiationVNumber >> ININOTES;
        cout << "Every " << irradiationPulse << "s, introducing " << irradiationINumber
<< " single I and " << irradiationVNumber << " single V." << endl;
        // Initial first event card
        // No need to decide where to insert, just get a card with type 20
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID = 0;
        EVENT_LIST[EVENT_INDEX].domainType = 3;
```

```
    EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
irradiationPulse;
    EVENT_LIST[EVENT_INDEX].eventType = 20;}
    fileInput2.close();
    // Sort event list
    for (i = 1; i < EVENT_INDEX;i++) {
    for (j = 1;j < EVENT_INDEX + 1 - i;j++) {
    if (EVENT_LIST[j].timestamp>EVENT_LIST[j + 1].timestamp) {
    swap(EVENT_LIST[j], EVENT_LIST[j + 1]);}}}
    // Update domain's card info
    for (i = 1;i < EVENT_INDEX;i++) {
        if (EVENT_LIST[i].domainType == 0) {
            HARD_PD[EVENT_LIST[i].domainID].event_card = i;}
        if (EVENT_LIST[i].domainType == 1) {
            SOFT_PD_A[EVENT_LIST[i].domainID].event_card = i;}}
    cout << endl;
    cout << "Finish Initialization." << endl;
    cout << "//////////////////////////////////////////////////////////////////////" << endl;
    // Event Loop
Test_Start_3:
    cout << "Entering main event stage." << endl;
    if (*getE10SolverVer - 0.9 < eps) {cout << "Current Version aCRD in (a) + (a) -
> (0) dynamics." << endl;}
    else {cout << "Current Version aCRD in (a) + (b) - > (a + b) dynamics." <<
endl;}
    // 1D prototype concerns a + a - > 0 annihilation
    // S2   -       E01     -       S0      -       Rebuild         -       S1/S2
    // Rebuild - E02 check
    // S0   -       E02     -       S0      -       Rebuild         -       S1/S2
    // S1   -       E10     -       Delete
    // 3D prototype with (a) + (b) - > (a + b) coalescence
    // S2   -       E01     -       S0      -       ASMS1(E02/E10 Update)
    // S2   -       E02     -       S0      -       ASMS1(E02/E10 Update)
    // S1   -       E10     -       S0      -       ASMS1(E02/E10 Update)
    // S0   -       ASMS2           -       S1/S2
    // Require full solver
    // Make sure identify all walkers which change their statue. No S0 is allowed
after event creator.
    STAGE_FLAG = 1; // Manual
    CONTROL_FLAG = 0;
    EVENT_POINTER = 1;
    // 1 Dimension begins here
    // No further upgrade plan
    if (space_dimension == 1) {
```

```
while (STAGE_FLAG){
cout << "Running 1 dimension events." << endl;
if (WORLD_CLOCK - END_CLOCK < eps) {
CONTROL_FLAG = 1; // Check with clock }
if (EVENT_POINTER > EVENT_INDEX) {
CONTROL_FLAG = 0; // All events finished}
while (CONTROL_FLAG){
// Begin to execute event
// Read event card
// Extra judge with E20 Event: TODO
currentEvent::domain_type = EVENT_LIST[EVENT_POINTER].domainType;
currentEvent::domainID = EVENT_LIST[EVENT_POINTER].domainID;
currentEvent::timestamp = EVENT_LIST[EVENT_POINTER].timestamp;
WORLD_CLOCK = currentEvent::timestamp;
// Execute E20 first, if not E20, do the rest
// Locate domain
if (currentEvent::domain_type == 0) {
// HPD event
currentEvent::event_type = HARD_PD[currentEvent::domainID].events_type;
// HPD concern E01 and E11, only E01 for this version
// May be cancelled by other event
if (currentEvent::event_type == -1) {
EVENT_POINTER = EVENT_POINTER + 1;}
if (currentEvent::event_type == 1) {
// Break HPD
// Disable the domain
HARD_PD[currentEvent::domainID].events_type = -1;
// Call solver to get location, set walker to S0, send walker
// getMajorDiffRelativeLocation1D(double Dv, double timestamp, double domainRadius, double walkerRadius, location centerLocation);
currentEvent::afterward_location = getMajorDiffRelativeLocation1D(ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].diffusivity, HARD_PD[currentEvent::domainID].events_time, HARD_PD[currentEvent::domainID].radius, ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].radius, HARD_PD[currentEvent::domainID].center);
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].statue = 0; // S0 release
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].center = currentEvent::afterward_location; // Move walker
// Move EVENT_POINTER to next event
EVENT_POINTER = EVENT_POINTER + 1;
// Enter ASMS 1st Stage
// 1. Check location and neighbour, ensure all S0 is away from HPD
```

```
        S0_INDEX = 1;
        S0_EVENT[1] = HARD_PD[currentEvent::domainID].walker_ID;
        S0_FULL_FLAG = 1;
        while (S0_FULL_FLAG){
        for (i = 0;i < S0_INDEX;i++) {
        S0_FULL_FLAG = 0;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        if (getHardDomainSurfaceDistance1D(ORI_WALKERS[S0_EVENT[i + 1]],
HARD_PD[j + 1]) - THRES_1 < eps) {
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].center =
getTransDiffRelativeLocation1D(ORI_WALKERS[HARD_PD[j +
1].walker_ID].diffusivity, WORLD_CLOCK - ORI_WALKERS[HARD_PD[j +
1].walker_ID].timestamp, HARD_PD[j + 1].radius, ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius, HARD_PD[j + 1].center);
        HARD_PD[j + 1].events_type = -1;
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].statue = 0;
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = HARD_PD[j + 1].walker_ID;
        S0_FULL_FLAG = 1; }}}}}
        cout << "Overall " << S0_INDEX << " Walkers Released in ASMS 1st Stage"
<< endl;
        // 1.5 Boundary Check for all S0 walkers
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.x +
ORI_WALKERS[S0_EVENT[i + 1]].radius - X_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.x - ORI_WALKERS[S0_EVENT[i +
1]].radius - X_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1;
        cout << "1 walker exits the boundary" << endl; }}}
        // Enter ASMS 2nd Stage
        // 2. All walkers rebuild
        if (S0_INDEX == 1) {
        if (ORI_WALKERS[S0_EVENT[1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance1D(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i
+ 1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[i + 1].current_member = SOFT_PD_A[i + 1].current_member + 1;
        SOFT_PD_A[i + 1].member_list.member[SOFT_PD_A[i + 1].current_member]
= ORI_WALKERS[S0_EVENT[1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time;
        SOFT_PD_A[i + 1].events_time = SOFT_PD_A[i + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[1]].statue = 1;
        ORI_WALKERS[S0_EVENT[1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1/1 Walker enters SPDa" << endl; }}}
        if (currentEvent::flag_s0_in_SPDa != 1) {
        currentEvent::min_distance = 999999999.99;
        currentEvent::near_domain_ID = 0;
        currentEvent::near_domain_type = 0;
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        if (HARD_PD[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getHardDomainSurfaceDistance1D(ORI_WALKERS[S0_EVENT[1]], HARD_PD[i +
1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1; }}}
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance1D(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i
+ 1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1;
        currentEvent::near_domain_type = 1; }}}
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x =
ORI_WALKERS[S0_EVENT[1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 0.0;
        HARD_PD[HPDOMAIN_INDEX].center.z = 0.0;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::min_distance;
        HARD_PD[HPDOMAIN_INDEX].walker_ID =
ORI_WALKERS[S0_EVENT[1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[1]].type;
        ORI_WALKERS[S0_EVENT[1]].statue = 2;
        ORI_WALKERS[S0_EVENT[1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1/1 Walker has been protected in HPD" << endl;}}}
        else {
        // 2.1 Whether this S0 walker located in SPDa
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance1D(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[j
+ 1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[j + 1].current_member = SOFT_PD_A[j + 1].current_member + 1;
        SOFT_PD_A[j + 1].member_list.member[SOFT_PD_A[j + 1].current_member]
= ORI_WALKERS[S0_EVENT[i + 1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time;
        SOFT_PD_A[j + 1].events_time = SOFT_PD_A[j + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1 Walker enters SPDa" << endl;}}}}}
        // 2.2 Whether any of other S0 walkers in S0_EVENT within TH2 to have SPDa
set up
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance1D(ORI_WALKERS[S0_EVENT[i
+ 1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i +
1]].radius - ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - THRES_2 < eps) {
        cout << "Create a new Type 1 SPD during this event." << endl;
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        SOFT_PD_A[SPDOMAINA_INDEX].ID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].center_distance =
currentEvent::walker_distance;
        SOFT_PD_A[SPDOMAINA_INDEX].clock = WORLD_CLOCK;
        SOFT_PD_A[SPDOMAINA_INDEX].events_type = 10;
        SOFT_PD_A[SPDOMAINA_INDEX].current_member = 2;
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[1] =
ORI_WALKERS[S0_EVENT[i + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[2] =
ORI_WALKERS[S0_EVENT[j + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].center1.x =
ORI_WALKERS[S0_EVENT[i + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.y =
ORI_WALKERS[S0_EVENT[i + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.z =
ORI_WALKERS[S0_EVENT[i + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.r =
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.x =
ORI_WALKERS[S0_EVENT[j + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.y =
ORI_WALKERS[S0_EVENT[j + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.z =
ORI_WALKERS[S0_EVENT[j + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.r =
ORI_WALKERS[S0_EVENT[j + 1]].radius;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[j + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = SPDOMAINA_INDEX;
        ORI_WALKERS[S0_EVENT[j + 1]].regionID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].events_time =
getReactionTimeStamp1D(SOFT_PD_A[SPDOMAINA_INDEX].member_list,
INITIAL_EVENTS_TIME);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
SOFT_PD_A[SPDOMAINA_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 1;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
SOFT_PD_A[SPDOMAINA_INDEX].events_time;
        SOFT_PD_A[SPDOMAINA_INDEX].event_card = EVENT_INDEX;
        cout << "2 walkers built a new SPDa" << endl;
        cout << ORI_WALKERS[S0_EVENT[i + 1]].ID << " and " <<
ORI_WALKERS[S0_EVENT[j + 1]].ID << endl; }}}}}
        // 2.3 For the rest of walkers ORI_WALKERS[S0_EVENT[i + 1]].statue == 0,
find the rest
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::O1_allow_r = 0.0;
        currentEvent::O1_neighbout_ID = 0;
        currentEvent::O2_allow_r = 0.0;
        currentEvent::O3_allow_r = 0.0;
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[j + 1]].type != -1 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance1D(ORI_WALKERS[S0_EVENT[i
+ 1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i +
1]].radius - ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O1_neighbout_ID = S0_EVENT[j + 1];
        currentEvent::O1_allow_r = getDistance1D(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]])*(ORI_WALKERS[S0_EVENT[i +
1]].diffusivity) / (ORI_WALKERS[S0_EVENT[i + 1]].diffusivity +
ORI_WALKERS[S0_EVENT[j + 1]].diffusivity) - ORI_WALKERS[S0_EVENT[i +
1]].radius;}}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
```

```
        currentEvent::walker_distance =
getHardDomainSurfaceDistance1D(ORI_WALKERS[S0_EVENT[i + 1]], HARD_PD[j
+ 1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O2_allow_r = currentEvent::walker_distance; }}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainACentertoCenterDistance1D(ORI_WALKERS[S0_EVENT[i + 1]],
SOFT_PD_A[j + 1]) - ORI_WALKERS[S0_EVENT[i + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O3_allow_r = currentEvent::walker_distance; }}}
        if (currentEvent::O1_allow_r - currentEvent::O2_allow_r < eps &&
currentEvent::O1_allow_r > eps) {
        if (currentEvent::O1_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 0.0;
        HARD_PD[HPDOMAIN_INDEX].center.z = 0.0;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O1_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 0.0;
        HARD_PD[HPDOMAIN_INDEX].center.z = 0.0;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}
        else {
        if (currentEvent::O2_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 0.0;
        HARD_PD[HPDOMAIN_INDEX].center.z = 0.0;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O2_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 0.0;
        HARD_PD[HPDOMAIN_INDEX].center.z = 0.0;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp1D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}}}}}
        if (currentEvent::event_type == 11) {
        // Move EVENT_POINTER to next event
        EVENT_POINTER = EVENT_POINTER + 1;
        cout << "FATAL ERROR 700: Unable to process E11 for HPD in this version.
Exit." << endl;
        system("pause");
        exit(0);}}
        if (currentEvent::domain_type == 1) {
currentEvent::event_type = SOFT_PD_A[currentEvent::domainID].events_type;
        // SPD concern E10 only, this is not in input of update
        // Only annihilation in this version,
        // Move EVENT_POINTER to next event
        EVENT_POINTER = EVENT_POINTER + 1;
        // Break SPDa
        SOFT_PD_A[currentEvent::domainID].events_type = -1;
        // Call solver to get location, send walker, set walker to S0 (No need)
        Update clock
        // Disable all walker (type to -1)
        for (i = 0;i < SOFT_PD_A[currentEvent::domainID].current_member;i++) {
        ORI_WALKERS[SOFT_PD_A[currentEvent::domainID].member_list.member[i
+ 1].ID].statue = 0;
        ORI_WALKERS[SOFT_PD_A[currentEvent::domainID].member_list.member[i
+ 1].ID].type = -1;}
        WORLD_CLOCK = currentEvent::timestamp;}
        if (currentEvent::domain_type == 2) {
        // Move EVENT_POINTER to next event
        EVENT_POINTER = EVENT_POINTER + 1;
        cout << "FATAL ERROR 700: Unable to process Type 2 SPD in this version.
Exit." << endl;
        system("pause");
        exit(0);}
        // Run event list sort
        for (i = 1; i < EVENT_INDEX;i++) {
        for (j = 1;j < EVENT_INDEX + 1 - i;j++) {
        if (EVENT_LIST[j].timestamp>EVENT_LIST[j + 1].timestamp) {
        swap(EVENT_LIST[j], EVENT_LIST[j + 1]);}}}
        for (i = 1;i < EVENT_INDEX;i++) {
```

```
        if (EVENT_LIST[i].domainType == 0) {
        HARD_PD[EVENT_LIST[i].domainID].event_card = i;}
        if (EVENT_LIST[i].domainType == 1) {
        SOFT_PD_A[EVENT_LIST[i].domainID].event_card = i; }}
        // One event has been processed with its executor and creator
        // Check the clock
        if (WORLD_CLOCK - END_CLOCK > eps) {
        CONTROL_FLAG = 0; // Check with clock}
        // Inner Dump Debug
        if (DUMPFLAG_INN) {
        ofstream file2("InnerDump.txt", ios::app);
        streambuf *f2 = cout.rdbuf(file2.rdbuf());
        DUMP_COUNTER = 0;
        cout << WORLD_CLOCK << endl;
        for (i = 1;i < WALKER_INDEX + 1;i++) {
        if (ORI_WALKERS[i].type != -1) {
        DUMP_COUNTER = DUMP_COUNTER + 1;
        if (space_dimension == 3) {
        cout << ORI_WALKERS[i].type << " " << ORI_WALKERS[i].component << "
" << ORI_WALKERS[i].center.x << " " << ORI_WALKERS[i].center.y << " " <<
ORI_WALKERS[i].center.z << endl;}
        if (space_dimension == 1) {
        cout << ORI_WALKERS[i].ID << " " << ORI_WALKERS[i].statue << " " <<
ORI_WALKERS[i].center.x << " 0.0 0.0" << endl;
        if (ORI_WALKERS[i].center.x - dump_min < eps) {
        dump_min = ORI_WALKERS[i].center.x;}
        if (ORI_WALKERS[i].center.x - dump_max > eps) {
        dump_max = ORI_WALKERS[i].center.x; }}}}
        cout << endl;
        cout.rdbuf(f2); }
        ofstream file3("timelog.txt", ios::app);
        streambuf *f3 = cout.rdbuf(file3.rdbuf());
        cout << WORLD_CLOCK << " " << DUMP_COUNTER << " " << dump_min
<< " " << dump_max << endl;
        cout.rdbuf(f3); }
        STAGE_FLAG = 0;
        cout << "Current calculation has finished at " << WORLD_CLOCK << "s,
aCRD has reached the end of clock, extend the calculation? 1:0." << endl;
        cin >> INPUT_FLAG;
        if (INPUT_FLAG) {
        cout << "Please input the extension time in the unit of second. " << endl;
        cin >> EXTRA_CLOCK;
        END_CLOCK = END_CLOCK + EXTRA_CLOCK;
        STAGE_FLAG = 1; }}}
        // 3 Dimension begins here
        if (space_dimension == 3) {
        while (STAGE_FLAG) {
        cout << "Running 3 dimension events." << endl;
        if (WORLD_CLOCK - END_CLOCK < eps) {
        CONTROL_FLAG = 1; // Check with clock}
        if (EVENT_POINTER > EVENT_INDEX) {
        CONTROL_FLAG = 0; // All events finished}
        while (CONTROL_FLAG) {
        // Begin to execute event
        // Read event card
        // Extra judge with E20 Event
        if (DISPLAYFLAG_SYS == 1) {cout << "Begin a new event." << endl;}
        currentEvent::domain_type = EVENT_LIST[EVENT_POINTER].domainType;
        currentEvent::domainID = EVENT_LIST[EVENT_POINTER].domainID;
        currentEvent::timestamp = EVENT_LIST[EVENT_POINTER].timestamp;
        currentEvent::event_type = EVENT_LIST[EVENT_POINTER].eventType;
        WORLD_CLOCK = currentEvent::timestamp;
        // Execute E20 first, if not E20, do the rest
        if (currentEvent::event_type == 20) {
        if (DISPLAYFLAG_SYS == 1) {cout << "Begin E20 event." << endl;}
        EVENT_POINTER = EVENT_POINTER + 1;
        // Create new walkers at a point out of HPD, can be in SPD
        // New walkers all in S0
        // Update WALKER_COUNTER and S0 log
        S0_INDEX = 0;
        for (i = 0;i < irradiationINumber;i++) {
        WALKER_COUNTER = WALKER_COUNTER + 1;
        ORI_WALKERS[WALKER_COUNTER].ID = WALKER_COUNTER;
        ORI_WALKERS[WALKER_COUNTER].timestamp = WORLD_CLOCK;
        ORI_WALKERS[WALKER_COUNTER].type = 1;
        ORI_WALKERS[WALKER_COUNTER].component = 1;
        ORI_WALKERS[WALKER_COUNTER].statue = 0;
        ORI_WALKERS[WALKER_COUNTER].energy = 0.0;
        ORI_WALKERS[WALKER_COUNTER].radius =
getEstimateRadius(ORI_WALKERS[WALKER_COUNTER]);
        ORI_WALKERS[WALKER_COUNTER].diffusivity =
getDvValue(ORI_WALKERS[WALKER_COUNTER].energy,
ORI_WALKERS[WALKER_COUNTER].type,
ORI_WALKERS[WALKER_COUNTER].component);
        // Out of HPD, can be in SPD
        // Search all walkers statue 0 or 1, type != -1, prevent overlap
        // Bypass ID=WALKER_COUNTER, this is yourself
        LOCATE_FLAG = 1;
        while (LOCATE_FLAG) {
        // Default regard as no overlap, may change during check
        LOCATE_FLAG = 0;
        // Get random location
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.x = X_LOWWER_LIMIT +
(X_UPPER_LIMIT - X_LOWWER_LIMIT)*RANDOM_SEED;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.y = Y_LOWWER_LIMIT +
(Y_UPPER_LIMIT - Y_LOWWER_LIMIT)*RANDOM_SEED;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.z = Z_LOWWER_LIMIT +
(Z_UPPER_LIMIT - Z_LOWWER_LIMIT)*RANDOM_SEED;
        // Begin check, if overlap, LOCATE_FLAG=1
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        // check HPD, no overlap
        if (HARD_PD[j + 1].events_type != -1) {
        new_walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[WALKER_COUNTER],
HARD_PD[j + 1]);
        if (new_walker_distance < eps) {
        LOCATE_FLAG = 1;}}}
        for (j = 1;j < WALKER_COUNTER;j++) {
        // check walkers with S0 and S1, no overlap
        if (ORI_WALKERS[j].type != -1) {
        if (ORI_WALKERS[j].statue == 1 || ORI_WALKERS[j].statue == 0) {
        new_walker_distance = getDistance(ORI_WALKERS[WALKER_COUNTER],
ORI_WALKERS[j]) - ORI_WALKERS[WALKER_COUNTER].radius -
ORI_WALKERS[j].radius;
        if (new_walker_distance < eps) {
        LOCATE_FLAG = 1;}}}}}}// Get good location
        // Update S0 log
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = ORI_WALKERS[WALKER_COUNTER].ID; // Log
the IDs of all walkers with S0} // End of I
        for (i = 0;i < irradiationVNumber;i++) {
        if (DISPLAYFLAG_SYS) {cout << "Create 1 new walker in this process." <<
endl; }
        WALKER_COUNTER = WALKER_COUNTER + 1;
        ORI_WALKERS[WALKER_COUNTER].ID = WALKER_COUNTER;
        ORI_WALKERS[WALKER_COUNTER].timestamp = WORLD_CLOCK;
        ORI_WALKERS[WALKER_COUNTER].type = 0;
        ORI_WALKERS[WALKER_COUNTER].component = 1;
        ORI_WALKERS[WALKER_COUNTER].statue = 0;
        ORI_WALKERS[WALKER_COUNTER].energy = 0.0;
        ORI_WALKERS[WALKER_COUNTER].radius =
getEstimateRadius(ORI_WALKERS[WALKER_COUNTER]);
        ORI_WALKERS[WALKER_COUNTER].diffusivity =
getDvValue(ORI_WALKERS[WALKER_COUNTER].energy,
ORI_WALKERS[WALKER_COUNTER].type,
ORI_WALKERS[WALKER_COUNTER].component);
        // Out of HPD, can be in SPD
        // Search all walkers statue 0 or 1, type != -1, prevent overlap
        // Bypass ID=WALKER_COUNTER, this is yourself
        LOCATE_FLAG = 1;
        while (LOCATE_FLAG) {
        // Default regard as no overlap, may change during check
        LOCATE_FLAG = 0;
        // Get random location
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.x = X_LOWWER_LIMIT +
(X_UPPER_LIMIT - X_LOWWER_LIMIT)*RANDOM_SEED;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.y = Y_LOWWER_LIMIT +
(Y_UPPER_LIMIT - Y_LOWWER_LIMIT)*RANDOM_SEED;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        ORI_WALKERS[WALKER_COUNTER].center.z = Z_LOWWER_LIMIT +
(Z_UPPER_LIMIT - Z_LOWWER_LIMIT)*RANDOM_SEED;
        // Begin check, if overlap, LOCATE_FLAG=1
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        // check HPD, no overlap
        if (HARD_PD[j + 1].events_type != -1) {
        new_walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[WALKER_COUNTER],
HARD_PD[j + 1]);
        if (new_walker_distance < eps) {
        LOCATE_FLAG = 1; }}}
        for (j = 1;j < WALKER_COUNTER;j++) {
        // check walkers with S0 and S1, no overlap
        if (ORI_WALKERS[j].type != -1) {
        if (ORI_WALKERS[j].statue == 1 || ORI_WALKERS[j].statue == 0) {
        new_walker_distance = getDistance(ORI_WALKERS[WALKER_COUNTER],
ORI_WALKERS[j]) - ORI_WALKERS[WALKER_COUNTER].radius -
ORI_WALKERS[j].radius;
        if (new_walker_distance < eps) {
        LOCATE_FLAG = 1;}}}}} // End of check}// Get good location
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = ORI_WALKERS[WALKER_COUNTER].ID; // Log
the IDs of all walkers with S0} // End of V
         // End of create walkers
        // Enter ASMS 1st Stage
        // 1. Check location and neighbour, E02 interface, ensure all S0 is away from
HPD
        S0_FULL_FLAG = 1;
        while (S0_FULL_FLAG) {
        for (i = 0;i < S0_INDEX;i++) {
        S0_FULL_FLAG = 0;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        if (getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]],
HARD_PD[j + 1]) - THRES_1 < eps) {
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].center =
getTransDiffRelativeLocation3D(ORI_WALKERS[HARD_PD[j +
1].walker_ID].diffusivity, WORLD_CLOCK - ORI_WALKERS[HARD_PD[j +
1].walker_ID].timestamp, HARD_PD[j + 1].radius, ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius, HARD_PD[j + 1].center);
        HARD_PD[j + 1].events_type = -1;
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].statue = 0;
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = HARD_PD[j + 1].walker_ID;
        S0_FULL_FLAG = 1; }}}}}
        cout << "Overall " << S0_INDEX << " Walkers Released in ASMS 1st Stage"
<< endl;
        // 1.5 Boundary Check for all S0 walkers
        for (i = 0;i < S0_INDEX;i++) {
        // X boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.x +
ORI_WALKERS[S0_EVENT[i + 1]].radius - X_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.x - ORI_WALKERS[S0_EVENT[i +
1]].radius - X_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle
        cout << "1 walker exits the X-axis boundary" << endl;}}
        // Y boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.y +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Y_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.y - ORI_WALKERS[S0_EVENT[i +
1]].radius - Y_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle
        cout << "1 walker exits the Y-axis boundary" << endl;}}
        // Z boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.z +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Z_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.z - ORI_WALKERS[S0_EVENT[i +
1]].radius - Z_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle
        cout << "1 walker exits the Z-axis boundary" << endl;}}}
        // Enter ASMS 2nd Stage
        // 2. All walkers rebuild
        if (S0_INDEX == 1) {
        if (ORI_WALKERS[S0_EVENT[1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[i + 1].current_member = SOFT_PD_A[i + 1].current_member + 1;
        SOFT_PD_A[i + 1].member_list.member[SOFT_PD_A[i + 1].current_member]
= ORI_WALKERS[S0_EVENT[1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time;
        SOFT_PD_A[i + 1].events_time = SOFT_PD_A[i + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[1]].statue = 1;
        ORI_WALKERS[S0_EVENT[1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1/1 Walker enters SPDa" << endl;}}}
        if (currentEvent::flag_s0_in_SPDa != 1) {
        currentEvent::min_distance = EXTRA_LARGE;
        currentEvent::near_domain_ID = 0;
        currentEvent::near_domain_type = 0;
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        if (HARD_PD[i + 1].events_type != -1) {
```

```
        currentEvent::walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[1]], HARD_PD[i +
1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1; }}}
        // Check all enabled SPDa
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1;
        currentEvent::near_domain_type = 1; }}}
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x =
ORI_WALKERS[S0_EVENT[1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y =
ORI_WALKERS[S0_EVENT[1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z =
ORI_WALKERS[S0_EVENT[1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::min_distance +
ORI_WALKERS[S0_EVENT[1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID =
ORI_WALKERS[S0_EVENT[1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[1]].type;
        ORI_WALKERS[S0_EVENT[1]].statue = 2;
        ORI_WALKERS[S0_EVENT[1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1/1 Walker has been protected in HPD" << endl;}}}
        else {
        // For all S0 walkers in original S0Index
        // 2.1 Whether this S0 walker located in SPDa
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[j +
1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[j + 1].current_member = SOFT_PD_A[j + 1].current_member + 1;
        SOFT_PD_A[j + 1].member_list.member[SOFT_PD_A[j + 1].current_member]
= ORI_WALKERS[S0_EVENT[i + 1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time;
        SOFT_PD_A[j + 1].events_time = SOFT_PD_A[j + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1 Walker enters SPDa" << endl;}}}}}
        // S0 not in SPDa
        // 2.2 Whether any of other S0 walkers in S0_EVENT within TH2 to have SPDa
set up
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
```

```
        if (currentEvent::walker_distance - THRES_2 < eps) {
        cout << "Create a new Type 1 SPD during this event." << endl;
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        SOFT_PD_A[SPDOMAINA_INDEX].ID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].center_distance =
currentEvent::walker_distance;
        SOFT_PD_A[SPDOMAINA_INDEX].clock = WORLD_CLOCK;
        SOFT_PD_A[SPDOMAINA_INDEX].events_type = 10;
        SOFT_PD_A[SPDOMAINA_INDEX].current_member = 2;
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[1] =
ORI_WALKERS[S0_EVENT[i + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[2] =
ORI_WALKERS[S0_EVENT[j + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].center1.x =
ORI_WALKERS[S0_EVENT[i + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.y =
ORI_WALKERS[S0_EVENT[i + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.z =
ORI_WALKERS[S0_EVENT[i + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.r =
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.x =
ORI_WALKERS[S0_EVENT[j + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.y =
ORI_WALKERS[S0_EVENT[j + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.z =
ORI_WALKERS[S0_EVENT[j + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.r =
ORI_WALKERS[S0_EVENT[j + 1]].radius;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[j + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = SPDOMAINA_INDEX;
        ORI_WALKERS[S0_EVENT[j + 1]].regionID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].events_time =
getReactionTimeStamp3D(SOFT_PD_A[SPDOMAINA_INDEX].member_list,
INITIAL_EVENTS_TIME);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
SOFT_PD_A[SPDOMAINA_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 1;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
SOFT_PD_A[SPDOMAINA_INDEX].events_time;
        SOFT_PD_A[SPDOMAINA_INDEX].event_card = EVENT_INDEX;
        cout << "2 walkers built a new SPDa" << endl;
        cout << ORI_WALKERS[S0_EVENT[i + 1]].ID << " and " <<
ORI_WALKERS[S0_EVENT[j + 1]].ID << endl;}}}}}
        // 2.3 For the rest of walkers ORI_WALKERS[S0_EVENT[i + 1]].statue == 0,
find the rest
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::O1_allow_r = 0.0;
        currentEvent::O1_neighbout_ID = 0;
        currentEvent::O2_allow_r = 0.0;
        currentEvent::O3_allow_r = 0.0;
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[j + 1]].type != -1 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O1_neighbout_ID = S0_EVENT[j + 1];
        currentEvent::O1_allow_r = getDistance(ORI_WALKERS[S0_EVENT[i + 1]],
ORI_WALKERS[S0_EVENT[j + 1]])*(ORI_WALKERS[S0_EVENT[i +
1]].diffusivity) / (ORI_WALKERS[S0_EVENT[i + 1]].diffusivity +
ORI_WALKERS[S0_EVENT[j + 1]].diffusivity) - ORI_WALKERS[S0_EVENT[i +
1]].radius;}}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]], HARD_PD[j
+ 1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O2_allow_r = currentEvent::walker_distance; }}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainACentertoCenterDistance(ORI_WALKERS[S0_EVENT[i + 1]],
SOFT_PD_A[j + 1]) - ORI_WALKERS[S0_EVENT[i + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O3_allow_r = currentEvent::walker_distance; }}}
        if (currentEvent::O1_allow_r - currentEvent::O2_allow_r < eps &&
currentEvent::O1_allow_r > eps) {
        if (currentEvent::O1_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O1_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
```

```
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}
        else {
        if (currentEvent::O2_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O2_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
```

```
          ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
          HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
          EVENT_INDEX = EVENT_INDEX + 1;
          EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
          EVENT_LIST[EVENT_INDEX].domainType = 0;
          EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
          HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
          cout << "1 Walker has been protected in HPD" << endl;}
          else {
          cout << "Create a new HPD during this event." << endl;
          HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
          HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
          HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
          HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
          HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
          HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
          HARD_PD[HPDOMAIN_INDEX].events_type = 1;
          HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
          HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
          HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
          ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
          ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
          HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
          EVENT_INDEX = EVENT_INDEX + 1;
          EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
          EVENT_LIST[EVENT_INDEX].domainType = 0;
          EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
          HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
```

```
        cout << "1 Walker has been protected in HPD" << endl;}}}}}
        // Create an extra new E20 card
        // No need to decide where to insert, just get a card with type 20
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID = 0;
        EVENT_LIST[EVENT_INDEX].domainType = 3;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK + irradiationPulse;
        EVENT_LIST[EVENT_INDEX].eventType = 20;}
        else {
        // Locate domain
        if (currentEvent::domain_type == 0) {
        // HPD event
        currentEvent::event_type = HARD_PD[currentEvent::domainID].events_type;
        // HPD concern E01 and E11, only E01 for this version
        // May be cancelled by other event
        if (currentEvent::event_type == -1) {
        EVENT_POINTER = EVENT_POINTER + 1;}
        if (currentEvent::event_type == 1) {
        // Break HPD
        HARD_PD[currentEvent::domainID].events_type = -1;
        // Call solver to get location, set walker to S0, send walker
        currentEvent::afterward_location = getMajorDiffRelativeLocation3D(ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].diffusivity, HARD_PD[currentEvent::domainID].events_time, HARD_PD[currentEvent::domainID].radius, ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].radius, HARD_PD[currentEvent::domainID].center);
        // Move EVENT_POINTER to next event
        EVENT_POINTER = EVENT_POINTER + 1;
        // Temp E11 starts from here
        if (getDissociationFlag(ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID])) {
        WALKER_COUNTER = WALKER_COUNTER + 1;
        ORI_WALKERS[WALKER_COUNTER].ID = WALKER_COUNTER;
        ORI_WALKERS[WALKER_COUNTER].timestamp = WORLD_CLOCK;
        ORI_WALKERS[WALKER_COUNTER].type = ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].type;
        ORI_WALKERS[WALKER_COUNTER].component = 1;
        ORI_WALKERS[WALKER_COUNTER].statue = 0;
        ORI_WALKERS[WALKER_COUNTER].energy = 0.0;
        ORI_WALKERS[WALKER_COUNTER].radius = getEstimateRadius(ORI_WALKERS[WALKER_COUNTER]);
        ORI_WALKERS[WALKER_COUNTER].diffusivity =
getDvValue(ORI_WALKERS[WALKER_COUNTER].energy,
ORI_WALKERS[WALKER_COUNTER].type,
ORI_WALKERS[WALKER_COUNTER].component);
        ORI_WALKERS[WALKER_COUNTER].center =
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].center;
        S0_INDEX = 1;
        S0_EVENT[S0_INDEX] = ORI_WALKERS[WALKER_COUNTER].ID;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].statue = 0;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].center =
currentEvent::afterward_location;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].component
= ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].component - 1;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].energy =
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].energy;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].radius =
getEstimateRadius(ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID])
;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].diffusivity =
getDvValue(ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].energy,
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].type,
ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].component);
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = HARD_PD[currentEvent::domainID].walker_ID; }
        else {
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].statue = 0;
        ORI_WALKERS[HARD_PD[currentEvent::domainID].walker_ID].center =
currentEvent::afterward_location;
        S0_INDEX = 1;
        S0_EVENT[1] = HARD_PD[currentEvent::domainID].walker_ID; }
        // Temp E11 ends here
        // Enter ASMS 1st Stage
        // 1. Check location and neighbour, E02 interface, ensure all S0 is away from
HPD
        S0_FULL_FLAG = 1;
        while (S0_FULL_FLAG) {
        // Search all exist S0 walkers
        for (i = 0;i < S0_INDEX;i++) {
        S0_FULL_FLAG = 0;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        if (getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]],
HARD_PD[j + 1]) - THRES_1 < eps) {
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].center =
getTransDiffRelativeLocation3D(ORI_WALKERS[HARD_PD[j +
1].walker_ID].diffusivity, WORLD_CLOCK - ORI_WALKERS[HARD_PD[j +
1].walker_ID].timestamp, HARD_PD[j + 1].radius, ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius, HARD_PD[j + 1].center);
        HARD_PD[j + 1].events_type = -1;
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].statue = 0;
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = HARD_PD[j + 1].walker_ID;
        S0_FULL_FLAG = 1;}}}}}
        cout << "Overall " << S0_INDEX << " Walkers Released in ASMS 1st Stage"
<< endl;
        // 1.5 Boundary Check for all S0 walkers
        for (i = 0;i < S0_INDEX;i++) {
        // X boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.x +
ORI_WALKERS[S0_EVENT[i + 1]].radius - X_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.x - ORI_WALKERS[S0_EVENT[i +
1]].radius - X_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}
        // Y boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.y +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Y_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.y - ORI_WALKERS[S0_EVENT[i +
1]].radius - Y_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}
        // Z boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.z +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Z_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.z - ORI_WALKERS[S0_EVENT[i +
1]].radius - Z_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}}
        // Enter ASMS 2nd Stage
        // 2. All walkers rebuild
        if (S0_INDEX == 1) {
        if (ORI_WALKERS[S0_EVENT[1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
```

```
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[i + 1].current_member = SOFT_PD_A[i + 1].current_member + 1;
        SOFT_PD_A[i + 1].member_list.member[SOFT_PD_A[i + 1].current_member]
= ORI_WALKERS[S0_EVENT[1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time;
        SOFT_PD_A[i + 1].events_time = SOFT_PD_A[i + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[1]].statue = 1;
        ORI_WALKERS[S0_EVENT[1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1/1 Walker enters SPDa" << endl;}}}
        // S0 not in SPDa
        if (currentEvent::flag_s0_in_SPDa != 1) {
        currentEvent::min_distance = EXTRA_LARGE;
        currentEvent::near_domain_ID = 0;
        currentEvent::near_domain_type = 0;
        for (i = 0;i < HPDOMAIN_INDEX;i++) {
        if (HARD_PD[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[1]], HARD_PD[i +
1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1;}}}
        for (i = 0;i < SPDOMAINA_INDEX;i++) {
        if (SOFT_PD_A[i + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::near_domain_ID = i + 1;
        currentEvent::near_domain_type = 1;}}}
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = 
ORI_WALKERS[S0_EVENT[1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = 
ORI_WALKERS[S0_EVENT[1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = 
ORI_WALKERS[S0_EVENT[1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::min_distance + 
ORI_WALKERS[S0_EVENT[1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = 
ORI_WALKERS[S0_EVENT[1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type = 
ORI_WALKERS[S0_EVENT[1]].type;
        ORI_WALKERS[S0_EVENT[1]].statue = 2;
        ORI_WALKERS[S0_EVENT[1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time = 
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius, 
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID = 
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK + 
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1/1 Walker has been protected in HPD" << endl;}}}
        else {
        // 2.1 Whether this S0 walker located in SPDa
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 && 
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance = 
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[j + 
1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[j + 1].current_member = SOFT_PD_A[j + 1].current_member + 1;
        SOFT_PD_A[j + 1].member_list.member[SOFT_PD_A[j + 1].current_member] 
= ORI_WALKERS[S0_EVENT[i + 1]];
```

```
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time;
        SOFT_PD_A[j + 1].events_time = SOFT_PD_A[j + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1 Walker enters SPDa" << endl;}}}}}
        // S0 not in SPDa
        // 2.2 Whether any of other S0 walkers in S0_EVENT within TH2 to have SPDa
set up
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - THRES_2 < eps) {
        cout << "Create a new Type 1 SPD during this event." << endl;
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        SOFT_PD_A[SPDOMAINA_INDEX].ID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].center_distance =
currentEvent::walker_distance;
        SOFT_PD_A[SPDOMAINA_INDEX].clock = WORLD_CLOCK;
        SOFT_PD_A[SPDOMAINA_INDEX].events_type = 10;
        SOFT_PD_A[SPDOMAINA_INDEX].current_member = 2;
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[1] =
ORI_WALKERS[S0_EVENT[i + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[2] =
ORI_WALKERS[S0_EVENT[j + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].center1.x =
ORI_WALKERS[S0_EVENT[i + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.y =
ORI_WALKERS[S0_EVENT[i + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.z =
ORI_WALKERS[S0_EVENT[i + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.r =
ORI_WALKERS[S0_EVENT[i + 1]].radius;
```

```
    SOFT_PD_A[SPDOMAINA_INDEX].center2.x =
ORI_WALKERS[S0_EVENT[j + 1]].center.x;
    SOFT_PD_A[SPDOMAINA_INDEX].center2.y =
ORI_WALKERS[S0_EVENT[j + 1]].center.y;
    SOFT_PD_A[SPDOMAINA_INDEX].center2.z =
ORI_WALKERS[S0_EVENT[j + 1]].center.z;
    SOFT_PD_A[SPDOMAINA_INDEX].center2.r =
ORI_WALKERS[S0_EVENT[j + 1]].radius;
    ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
    ORI_WALKERS[S0_EVENT[j + 1]].statue = 1;
    ORI_WALKERS[S0_EVENT[i + 1]].regionID = SPDOMAINA_INDEX;
    ORI_WALKERS[S0_EVENT[j + 1]].regionID = SPDOMAINA_INDEX;
    SOFT_PD_A[SPDOMAINA_INDEX].events_time =
getReactionTimeStamp3D(SOFT_PD_A[SPDOMAINA_INDEX].member_list,
INITIAL_EVENTS_TIME);
    EVENT_INDEX = EVENT_INDEX + 1;
    EVENT_LIST[EVENT_INDEX].domainID =
SOFT_PD_A[SPDOMAINA_INDEX].ID;
    EVENT_LIST[EVENT_INDEX].domainType = 1;
    EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
SOFT_PD_A[SPDOMAINA_INDEX].events_time;
    SOFT_PD_A[SPDOMAINA_INDEX].event_card = EVENT_INDEX;
    cout << "2 walkers built a new SPDa" << endl;
    cout << ORI_WALKERS[S0_EVENT[i + 1]].ID << " and " <<
ORI_WALKERS[S0_EVENT[j + 1]].ID << endl; }}}}}
    // 2.3 For the rest of walkers ORI_WALKERS[S0_EVENT[i + 1]].statue == 0,
find the rest
    for (i = 0;i < S0_INDEX;i++) {
    if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
    currentEvent::O1_allow_r = 0.0;
    currentEvent::O1_neighbout_ID = 0;
    currentEvent::O2_allow_r = 0.0;
    currentEvent::O3_allow_r = 0.0;
    currentEvent::min_distance = EXTRA_LARGE;
    for (j = i + 1;j < S0_INDEX;j++) {
    if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[j + 1]].type != -1 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
    currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
    if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
    currentEvent::min_distance = currentEvent::walker_distance;
```

```
        currentEvent::O1_neighbout_ID = S0_EVENT[j + 1];
        currentEvent::O1_allow_r = getDistance(ORI_WALKERS[S0_EVENT[i + 1]], ORI_WALKERS[S0_EVENT[j + 1]])*(ORI_WALKERS[S0_EVENT[i + 1]].diffusivity) / (ORI_WALKERS[S0_EVENT[i + 1]].diffusivity + ORI_WALKERS[S0_EVENT[j + 1]].diffusivity) - ORI_WALKERS[S0_EVENT[i + 1]].radius;}}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        currentEvent::walker_distance = getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]], HARD_PD[j + 1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O2_allow_r = currentEvent::walker_distance; }}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance = getSoftDomainACentertoCenterDistance(ORI_WALKERS[S0_EVENT[i + 1]], SOFT_PD_A[j + 1]) - ORI_WALKERS[S0_EVENT[i + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O3_allow_r = currentEvent::walker_distance;}}}
        if (currentEvent::O1_allow_r - currentEvent::O2_allow_r < eps && currentEvent::O1_allow_r > eps) {
        if (currentEvent::O1_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i + 1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i + 1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i + 1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O1_allow_r + ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i + 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type = ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
```

```
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}
        else {
        if (currentEvent::O2_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O2_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
```

```
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}}}}}
        if (currentEvent::event_type == 11) {
        EVENT_POINTER = EVENT_POINTER + 1;
        cout << "FATAL ERROR 700: Unable to process E11 for HPD in this version.
Exit." << endl;
        system("pause");
        exit(0);}}
        if (currentEvent::domain_type == 1) {
        // SPDa event
        currentEvent::event_type = SOFT_PD_A[currentEvent::domainID].events_type;
        // SPD concern E10 only, this is not in input of update
        // Coalescence enabled
        // Move EVENT_POINTER to next event
        EVENT_POINTER = EVENT_POINTER + 1;
        // Break SPDa
        SOFT_PD_A[currentEvent::domainID].events_type = -1;
        // Call solver to get location, send walker, set walker to S0
        // Disable all current walker (type to -1)
        for (i = 0;i < SOFT_PD_A[currentEvent::domainID].current_member;i++) {
        ORI_WALKERS[SOFT_PD_A[currentEvent::domainID].member_list.member[i
+ 1].ID].statue = 0;
```

```
          ORI_WALKERS[SOFT_PD_A[currentEvent::domainID].member_list.member[i
+ 1].ID].type = -1;}
          // Call Solver to find the find statue
          SOFT_PD_A[currentEvent::domainID].e10_final_statue =
getReactionFinalStatues3D(SOFT_PD_A[currentEvent::domainID].member_list,
SOFT_PD_A[currentEvent::domainID].current_member,
SOFT_PD_A[currentEvent::domainID].center1,
SOFT_PD_A[currentEvent::domainID].center2);
          SOFT_PD_A[currentEvent::domainID].final_member = 0;
          for (i = 1;i < 6;i++) {
          if (SOFT_PD_A[currentEvent::domainID].e10_final_statue.member[i].type != -
1) {
          SOFT_PD_A[currentEvent::domainID].final_member =
SOFT_PD_A[currentEvent::domainID].final_member + 1;}}
          cout << "Encount with " <<
SOFT_PD_A[currentEvent::domainID].final_member << " new walker(s) in this event."
<< endl;
          // Create new walkers, all in S0
          S0_INDEX = 0;
          for (i = 0;i < SOFT_PD_A[currentEvent::domainID].final_member;i++) {
          // Update walker counter, point to new walker
          WALKER_COUNTER = WALKER_COUNTER + 1;
          ORI_WALKERS[WALKER_COUNTER].ID = WALKER_COUNTER;
          ORI_WALKERS[WALKER_COUNTER].timestamp = WORLD_CLOCK;
          ORI_WALKERS[WALKER_COUNTER].type =
SOFT_PD_A[currentEvent::domainID].e10_final_statue.member[i + 1].type;
          ORI_WALKERS[WALKER_COUNTER].component =
SOFT_PD_A[currentEvent::domainID].e10_final_statue.member[i + 1].component;
          ORI_WALKERS[WALKER_COUNTER].statue = 0;
          ORI_WALKERS[WALKER_COUNTER].energy = 0.0;
          ORI_WALKERS[WALKER_COUNTER].center =
SOFT_PD_A[currentEvent::domainID].e10_final_statue.member[i + 1].center;
          ORI_WALKERS[WALKER_COUNTER].radius =
getEstimateRadius(ORI_WALKERS[WALKER_COUNTER]);
          ORI_WALKERS[WALKER_COUNTER].diffusivity =
getDvValue(ORI_WALKERS[WALKER_COUNTER].energy,
ORI_WALKERS[WALKER_COUNTER].type,
ORI_WALKERS[WALKER_COUNTER].component);
          S0_INDEX = S0_INDEX + 1;
          S0_EVENT[S0_INDEX] = ORI_WALKERS[WALKER_COUNTER].ID;}
          if (S0_INDEX == 0) {
          // No new walker, bypass ASMS
          cout << "Annihilation in SPDa." << endl;}
          else {
```

```
        // Begin ASMS, full check around
        // Don't touch EVENT_POINTER anymore
        // Enter ASMS 1st Stage
        // 1. Check location and neighbour, E02 interface, ensure all S0 is away from
HPD
        S0_FULL_FLAG = 1;
        while (S0_FULL_FLAG) {
        for (i = 0;i < S0_INDEX;i++) {
        S0_FULL_FLAG = 0;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        if (getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]],
HARD_PD[j + 1]) - THRES_1 < eps) {
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].center =
getTransDiffRelativeLocation3D(ORI_WALKERS[HARD_PD[j +
1].walker_ID].diffusivity, WORLD_CLOCK - ORI_WALKERS[HARD_PD[j +
1].walker_ID].timestamp, HARD_PD[j + 1].radius, ORI_WALKERS[HARD_PD[j +
1].walker_ID].radius, HARD_PD[j + 1].center);
        HARD_PD[j + 1].events_type = -1;
        ORI_WALKERS[HARD_PD[j + 1].walker_ID].statue = 0;
        S0_INDEX = S0_INDEX + 1;
        S0_EVENT[S0_INDEX] = HARD_PD[j + 1].walker_ID;
        S0_FULL_FLAG = 1;}}}}}
        cout << "Overall " << S0_INDEX << " Walkers Released in ASMS 1st Stage"
<< endl;
        // 1.5 Boundary Check for all S0 walkers
        for (i = 0;i < S0_INDEX;i++) {
        // X boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.x +
ORI_WALKERS[S0_EVENT[i + 1]].radius - X_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.x - ORI_WALKERS[S0_EVENT[i +
1]].radius - X_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}
        // Y boundary
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].center.y +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Y_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.y - ORI_WALKERS[S0_EVENT[i +
1]].radius - Y_LOWWER_LIMIT < eps) {
        ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}
        // Z boundary
```

```
	if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
	if (ORI_WALKERS[S0_EVENT[i + 1]].center.z +
ORI_WALKERS[S0_EVENT[i + 1]].radius - Z_UPPER_LIMIT>eps ||
ORI_WALKERS[S0_EVENT[i + 1]].center.z - ORI_WALKERS[S0_EVENT[i +
1]].radius - Z_LOWWER_LIMIT < eps) {
	ORI_WALKERS[S0_EVENT[i + 1]].type = -1; // Inable this particle}}}
	// Enter ASMS 2nd Stage
	// Protect walker again, build new domain or update SPDa
	// 2. All walkers rebuild
	if (S0_INDEX == 1) {
	if (ORI_WALKERS[S0_EVENT[1]].type != -1) {
	currentEvent::flag_s0_in_SPDa = 0;
	for (i = 0;i < SPDOMAINA_INDEX;i++) {
	if (SOFT_PD_A[i + 1].events_type != -1) {
	currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
	if (currentEvent::walker_distance - THRES_2 < eps) {
	currentEvent::flag_s0_in_SPDa = 1;
	SOFT_PD_A[i + 1].current_member = SOFT_PD_A[i + 1].current_member + 1;
	SOFT_PD_A[i + 1].member_list.member[SOFT_PD_A[i + 1].current_member]
= ORI_WALKERS[S0_EVENT[1]];
	currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time;
	SOFT_PD_A[i + 1].events_time = SOFT_PD_A[i + 1].events_time;
	currentEvent::update_spda_time_temp = SOFT_PD_A[i + 1].events_time -
currentEvent::update_spda_time_temp;
	ORI_WALKERS[S0_EVENT[1]].statue = 1;
	ORI_WALKERS[S0_EVENT[1]].timestamp = WORLD_CLOCK;
	EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[i + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
	cout << "1/1 Walker enters SPDa" << endl;}}}
	if (currentEvent::flag_s0_in_SPDa != 1) {
	currentEvent::min_distance = EXTRA_LARGE;
	currentEvent::near_domain_ID = 0;
	currentEvent::near_domain_type = 0;
	for (i = 0;i < HPDOMAIN_INDEX;i++) {
	if (HARD_PD[i + 1].events_type != -1) {
	currentEvent::walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[1]], HARD_PD[i +
1]);
	if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
	currentEvent::min_distance = currentEvent::walker_distance;
	currentEvent::near_domain_ID = i + 1;}}}
	for (i = 0;i < SPDOMAINA_INDEX;i++) {
	if (SOFT_PD_A[i + 1].events_type != -1) {
	currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[i +
1]);
	if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
	currentEvent::min_distance = currentEvent::walker_distance;
	currentEvent::near_domain_ID = i + 1;
	currentEvent::near_domain_type = 1;}}}
	cout << "Create a new HPD during this event." << endl;
	HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
	HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
	HARD_PD[HPDOMAIN_INDEX].center.x =
ORI_WALKERS[S0_EVENT[1]].center.x;
	HARD_PD[HPDOMAIN_INDEX].center.y =
ORI_WALKERS[S0_EVENT[1]].center.y;
	HARD_PD[HPDOMAIN_INDEX].center.z =
ORI_WALKERS[S0_EVENT[1]].center.z;
	HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
	HARD_PD[HPDOMAIN_INDEX].events_type = 1;
	HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::min_distance +
ORI_WALKERS[S0_EVENT[1]].radius;
	HARD_PD[HPDOMAIN_INDEX].walker_ID =
ORI_WALKERS[S0_EVENT[1]].ID;
	HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[1]].type;
	ORI_WALKERS[S0_EVENT[1]].statue = 2;
	ORI_WALKERS[S0_EVENT[1]].regionID = HPDOMAIN_INDEX;
	HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
	EVENT_INDEX = EVENT_INDEX + 1;
	EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
	EVENT_LIST[EVENT_INDEX].domainType = 0;
	EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
	HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
	cout << "1/1 Walker has been protected in HPD" << endl;}}}
	else {
	// 2.1 Whether this S0 walker located in SPDa
	for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::flag_s0_in_SPDa = 0;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainASurfaceDistance(ORI_WALKERS[S0_EVENT[1]], SOFT_PD_A[j +
1]);
        if (currentEvent::walker_distance - THRES_2 < eps) {
        currentEvent::flag_s0_in_SPDa = 1;
        SOFT_PD_A[j + 1].current_member = SOFT_PD_A[j + 1].current_member + 1;
        SOFT_PD_A[j + 1].member_list.member[SOFT_PD_A[j + 1].current_member]
= ORI_WALKERS[S0_EVENT[i + 1]];
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time;
        SOFT_PD_A[j + 1].events_time = SOFT_PD_A[j + 1].events_time;
        currentEvent::update_spda_time_temp = SOFT_PD_A[j + 1].events_time -
currentEvent::update_spda_time_temp;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].timestamp = WORLD_CLOCK;
        EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp =
EVENT_LIST[SOFT_PD_A[j + 1].event_card].timestamp +
currentEvent::update_spda_time_temp;
        cout << "1 Walker enters SPDa" << endl;}}}}}
        // 2.2 Whether any of other S0 walkers in S0_EVENT within TH2 to have SPDa
set up
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - THRES_2 < eps) {
        cout << "Create a new Type 1 SPD during this event." << endl;
        SPDOMAINA_INDEX = SPDOMAINA_INDEX + 1;
        SOFT_PD_A[SPDOMAINA_INDEX].ID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].center_distance =
currentEvent::walker_distance;
        SOFT_PD_A[SPDOMAINA_INDEX].clock = WORLD_CLOCK;
        SOFT_PD_A[SPDOMAINA_INDEX].events_type = 10;
        SOFT_PD_A[SPDOMAINA_INDEX].current_member = 2;
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[1] =
ORI_WALKERS[S0_EVENT[i + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].member_list.member[2] =
ORI_WALKERS[S0_EVENT[j + 1]];
        SOFT_PD_A[SPDOMAINA_INDEX].center1.x =
ORI_WALKERS[S0_EVENT[i + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.y =
ORI_WALKERS[S0_EVENT[i + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.z =
ORI_WALKERS[S0_EVENT[i + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center1.r =
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.x =
ORI_WALKERS[S0_EVENT[j + 1]].center.x;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.y =
ORI_WALKERS[S0_EVENT[j + 1]].center.y;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.z =
ORI_WALKERS[S0_EVENT[j + 1]].center.z;
        SOFT_PD_A[SPDOMAINA_INDEX].center2.r =
ORI_WALKERS[S0_EVENT[j + 1]].radius;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[j + 1]].statue = 1;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = SPDOMAINA_INDEX;
        ORI_WALKERS[S0_EVENT[j + 1]].regionID = SPDOMAINA_INDEX;
        SOFT_PD_A[SPDOMAINA_INDEX].events_time =
getReactionTimeStamp3D(SOFT_PD_A[SPDOMAINA_INDEX].member_list,
INITIAL_EVENTS_TIME);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
SOFT_PD_A[SPDOMAINA_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 1;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
SOFT_PD_A[SPDOMAINA_INDEX].events_time;
        SOFT_PD_A[SPDOMAINA_INDEX].event_card = EVENT_INDEX;
        cout << "2 walkers built a new SPDa" << endl;
        cout << ORI_WALKERS[S0_EVENT[i + 1]].ID << " and " <<
ORI_WALKERS[S0_EVENT[j + 1]].ID << endl;}}}}}
        // 2.3 For the rest of walkers ORI_WALKERS[S0_EVENT[i + 1]].statue == 0,
find the rest
        for (i = 0;i < S0_INDEX;i++) {
        if (ORI_WALKERS[S0_EVENT[i + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[i + 1]].type != -1) {
        currentEvent::O1_allow_r = 0.0;
        currentEvent::O1_neighbout_ID = 0;
        currentEvent::O2_allow_r = 0.0;
        currentEvent::O3_allow_r = 0.0;
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = i + 1;j < S0_INDEX;j++) {
        if (ORI_WALKERS[S0_EVENT[j + 1]].statue == 0 &&
ORI_WALKERS[S0_EVENT[j + 1]].type != -1 && ORI_WALKERS[S0_EVENT[j +
1]].type != -1) {
        currentEvent::walker_distance = getDistance(ORI_WALKERS[S0_EVENT[i +
1]], ORI_WALKERS[S0_EVENT[j + 1]]) - ORI_WALKERS[S0_EVENT[i + 1]].radius
- ORI_WALKERS[S0_EVENT[j + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O1_neighbout_ID = S0_EVENT[j + 1];
        currentEvent::O1_allow_r = getDistance(ORI_WALKERS[S0_EVENT[i + 1]],
ORI_WALKERS[S0_EVENT[j + 1]])*(ORI_WALKERS[S0_EVENT[i +
1]].diffusivity) / (ORI_WALKERS[S0_EVENT[i + 1]].diffusivity +
ORI_WALKERS[S0_EVENT[j + 1]].diffusivity) - ORI_WALKERS[S0_EVENT[i +
1]].radius;}}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < HPDOMAIN_INDEX;j++) {
        if (HARD_PD[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getHardDomainSurfaceDistance3D(ORI_WALKERS[S0_EVENT[i + 1]], HARD_PD[j
+ 1]);
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O2_allow_r = currentEvent::walker_distance; }}}
        currentEvent::min_distance = EXTRA_LARGE;
        for (j = 0;j < SPDOMAINA_INDEX;j++) {
        if (SOFT_PD_A[j + 1].events_type != -1) {
        currentEvent::walker_distance =
getSoftDomainACentertoCenterDistance(ORI_WALKERS[S0_EVENT[i + 1]],
SOFT_PD_A[j + 1]) - ORI_WALKERS[S0_EVENT[i + 1]].radius;
        if (currentEvent::walker_distance - currentEvent::min_distance < eps) {
        currentEvent::min_distance = currentEvent::walker_distance;
        currentEvent::O3_allow_r = currentEvent::walker_distance; }}}
        if (currentEvent::O1_allow_r - currentEvent::O2_allow_r < eps &&
currentEvent::O1_allow_r > eps) {
        if (currentEvent::O1_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
```

```
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O1_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
```

```
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}
        else {
        if (currentEvent::O2_allow_r - currentEvent::O3_allow_r < eps) {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O2_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}
        else {
        cout << "Create a new HPD during this event." << endl;
        HPDOMAIN_INDEX = HPDOMAIN_INDEX + 1;
        HARD_PD[HPDOMAIN_INDEX].ID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].center.x = ORI_WALKERS[S0_EVENT[i +
1]].center.x;
        HARD_PD[HPDOMAIN_INDEX].center.y = ORI_WALKERS[S0_EVENT[i +
1]].center.y;
        HARD_PD[HPDOMAIN_INDEX].center.z = ORI_WALKERS[S0_EVENT[i +
1]].center.z;
        HARD_PD[HPDOMAIN_INDEX].clock = WORLD_CLOCK;
        HARD_PD[HPDOMAIN_INDEX].events_type = 1;
        HARD_PD[HPDOMAIN_INDEX].radius = currentEvent::O3_allow_r +
ORI_WALKERS[S0_EVENT[i + 1]].radius;
        HARD_PD[HPDOMAIN_INDEX].walker_ID = ORI_WALKERS[S0_EVENT[i
+ 1]].ID;
        HARD_PD[HPDOMAIN_INDEX].walker_type =
ORI_WALKERS[S0_EVENT[i + 1]].type;
        ORI_WALKERS[S0_EVENT[i + 1]].statue = 2;
        ORI_WALKERS[S0_EVENT[i + 1]].regionID = HPDOMAIN_INDEX;
        HARD_PD[HPDOMAIN_INDEX].events_time =
getMajorDiffTimeStamp3D(ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walk
er_ID].diffusivity, HARD_PD[HPDOMAIN_INDEX].radius,
ORI_WALKERS[HARD_PD[HPDOMAIN_INDEX].walker_ID].radius);
        EVENT_INDEX = EVENT_INDEX + 1;
        EVENT_LIST[EVENT_INDEX].domainID =
HARD_PD[HPDOMAIN_INDEX].ID;
        EVENT_LIST[EVENT_INDEX].domainType = 0;
        EVENT_LIST[EVENT_INDEX].timestamp = WORLD_CLOCK +
HARD_PD[HPDOMAIN_INDEX].events_time;
        HARD_PD[HPDOMAIN_INDEX].event_card = EVENT_INDEX;
        cout << "1 Walker has been protected in HPD" << endl;}}}}}}}
        if (currentEvent::domain_type == 2) {
        EVENT_POINTER = EVENT_POINTER + 1;
        cout << "FATAL ERROR 700: Unable to process Type 2 SPD in this version.
Exit." << endl;
        system("pause");
        exit(0);}}
        // End of event
        // Run event list sort
```

```
          for (i = 1; i < EVENT_INDEX;i++) {
          for (j = 1;j < EVENT_INDEX + 1 - i;j++) {
          if (EVENT_LIST[j].timestamp>EVENT_LIST[j + 1].timestamp) {
          swap(EVENT_LIST[j], EVENT_LIST[j + 1]);}}}
          for (i = 1;i < EVENT_INDEX;i++) {
          if (EVENT_LIST[i].domainType == 0) {
          HARD_PD[EVENT_LIST[i].domainID].event_card = i;}
          if (EVENT_LIST[i].domainType == 1) {
          SOFT_PD_A[EVENT_LIST[i].domainID].event_card = i;}}
          // One event has been processed with its executor and creator
          // Check the clock
          if (WORLD_CLOCK - END_CLOCK > eps) {
          CONTROL_FLAG = 0; }
          // Inner Dump Debug
          if (DUMPFLAG_INN) {
          ofstream file2("InnerDump.txt", ios::app);
          streambuf *f2 = cout.rdbuf(file2.rdbuf());
          DUMP_COUNTER = 0;
          cout << WORLD_CLOCK << endl;
          for (i = 1;i < WALKER_INDEX + 1;i++) {
          if (ORI_WALKERS[i].type != -1) {
          DUMP_COUNTER = DUMP_COUNTER + 1;
          if (space_dimension == 3) {
          cout << " " << ORI_WALKERS[i].type << " " <<
ORI_WALKERS[i].component << " " << ORI_WALKERS[i].center.x << " " <<
ORI_WALKERS[i].center.y << " " << ORI_WALKERS[i].center.z << endl;}}}
          cout << endl;
          cout.rdbuf(f2);}
          // OS2
          ofstream file3("timelog.txt", ios::app);
          streambuf *f3 = cout.rdbuf(file3.rdbuf());
          cout << WORLD_CLOCK << " " << DUMP_COUNTER << endl;
          cout.rdbuf(f3);
          // OS3
          ofstream file4("Original_voids.txt", ios::app);
          streambuf *f4 = cout.rdbuf(file4.rdbuf());
          long dump_voids;
          dump_voids = 0;
          for (i = 1;i < WALKER_INDEX + 1;i++) {
          if (ORI_WALKERS[i].type == 0) {
          if (space_dimension == 3) {
          dump_voids = dump_voids + ORI_WALKERS[i].component;}}}
          cout << WORLD_CLOCK << " " << dump_voids << endl;
          cout.rdbuf(f4);
```

```
        // OS4
        ofstream file5("Total_voids.txt", ios::app);
        streambuf *f5 = cout.rdbuf(file5.rdbuf());
        dump_voids = 0;
        for (i = 1;i < WALKER_COUNTER + 1;i++) {
        if (ORI_WALKERS[i].type == 0) {
        if (space_dimension == 3) {
        dump_voids = dump_voids + ORI_WALKERS[i].component;}}}
        cout << WORLD_CLOCK << " " << dump_voids << endl;
        cout.rdbuf(f5);
        // OS5
        if (DUMPLAMMPSFLAG_INN) {
        DUMPCOUNTER_INN = DUMPCOUNTER_INN + 1;
        if (DUMPCOUNTER_INN == DUMPCOUNTER_FLAG) {
        DUMPCOUNTER_INN = 0;
        ofstream file6("LAMMPS_dump.txt", ios::app);
        streambuf *f6 = cout.rdbuf(file6.rdbuf());
        long lammps_type = 0;
        long lammps_number = 0;
        cout << "ITEM: TIMESTEP" << endl;
        //TIME_STEP = TIME_STEP + 1;
        cout << WORLD_CLOCK << endl;
        cout << "ITEM: NUMBER OF ATOMS" << endl;
        for (i = 1;i < WALKER_COUNTER + 1;i++) {
        if (ORI_WALKERS[i].type != -1) {
        lammps_number = lammps_number + 1;}}
        cout << lammps_number << endl;
        cout << "ITEM: BOX BOUNDS pp pp pp" << endl;
        cout << X_LOWWER_LIMIT << " " << X_UPPER_LIMIT << endl;
        cout << Y_LOWWER_LIMIT << " " << Y_UPPER_LIMIT << endl;
        cout << Z_LOWWER_LIMIT << " " << Z_UPPER_LIMIT << endl;
        cout << "ITEM: ATOMS id type x y z c_ke c_pe " << endl;
        for (i = 1;i < WALKER_COUNTER + 1;i++) {
        if (ORI_WALKERS[i].type != -1) {
        if (ORI_WALKERS[i].type == 0) {
        lammps_type = ORI_WALKERS[i].component;}
        if (ORI_WALKERS[i].type == 1) {
        lammps_type = ORI_WALKERS[i].component + 100;}
        cout << ORI_WALKERS[i].ID << " " << lammps_type << " " <<
ORI_WALKERS[i].center.x << " " << ORI_WALKERS[i].center.y << " " <<
ORI_WALKERS[i].center.z << " 0.0 " << ORI_WALKERS[i].energy << endl;}}
        cout.rdbuf(f6);}}
        // OS6
        // End of inner dump}
```

```
	STAGE_FLAG = 0;
	cout << "Current calculation has finished at " << WORLD_CLOCK << "s, 
aCRD has reached the end of clock, extend the calculation? 1:0." << endl;
	cin >> INPUT_FLAG;
	if (INPUT_FLAG) {
	cout << "Please input the extension time in the unit of second. " << endl;
	cin >> EXTRA_CLOCK;
	END_CLOCK = END_CLOCK + EXTRA_CLOCK;
	STAGE_FLAG = 1;}}}
	cout << "Finish Event Loop." << endl;
	cout << "////////////////////////////////////////////////////////////" << endl;
	// Final Sync
	cout << "Begin to do final synchronization." << endl;
	// Run E02 to all HPDs with
	// getSynchronization();
	// HPDs run E02
	// SPD run E10_END, in current version, do not handle SPDa as Trigger time is 
very short after SPDa built
	for (i = 0;i < WALKER_COUNTER;i++) {
	if (ORI_WALKERS[i + 1].statue == 2) {
	if (space_dimension == 1) {
	ORI_WALKERS[i + 1].statue = 0;
	ORI_WALKERS[i + 1].center = 
getTransDiffRelativeLocation1D(ORI_WALKERS[i + 1].diffusivity, WORLD_CLOCK 
- ORI_WALKERS[i + 1].timestamp, HARD_PD[ORI_WALKERS[i + 
1].regionID].radius, ORI_WALKERS[i + 1].radius, ORI_WALKERS[i + 1].center);}
	if (space_dimension == 3) {
	ORI_WALKERS[i + 1].statue = 0;
	ORI_WALKERS[i + 1].center = 
getTransDiffRelativeLocation3D(ORI_WALKERS[i + 1].diffusivity, WORLD_CLOCK 
- ORI_WALKERS[i + 1].timestamp, HARD_PD[ORI_WALKERS[i + 
1].regionID].radius, ORI_WALKERS[i + 1].radius, ORI_WALKERS[i + 1].center);}}
	if (ORI_WALKERS[i + 1].statue == 1) {
	if (space_dimension == 1) {
	for (j = 0;j < SOFT_PD_A[ORI_WALKERS[i + 
1].regionID].current_member;j++) {
	ORI_WALKERS[SOFT_PD_A[ORI_WALKERS[i + 
1].regionID].member_list.member[j + 1].ID].statue = 0;
	ORI_WALKERS[SOFT_PD_A[ORI_WALKERS[i + 
1].regionID].member_list.member[j + 1].ID].type = -1;}}
	if (space_dimension == 3) {// Don't do anything}}}
	cout << "Finish synchronization." << endl;
	cout << "////////////////////////////////////////////////////////////" << endl;
	// Debug Goto
	cout << "Test Again? 1 to Restart at File Loading, 2 to Restart at Introduce 
Solvers and Event Queue, 3 to Restart at Event Loop, 0 to continue." << endl;
	int Test_Flag;
	cin >> Test_Flag;
	if (Test_Flag == 1) goto Test_Start_1;
	if (Test_Flag == 2) goto Test_Start_2;
	if (Test_Flag == 3) goto Test_Start_3;
//Test_Start_4:
	ofstream file("Dump.txt", ios::app);
	streambuf *f = cout.rdbuf(file.rdbuf());
	cout << WORLD_CLOCK << endl;
	for (i = 1;i < WALKER_INDEX;i++) {
	if (ORI_WALKERS[i].type != -1) {
	if (space_dimension == 3) {
	cout << ORI_WALKERS[i].type << " " << ORI_WALKERS[i].component << " 
" << ORI_WALKERS[i].center.x << " " << ORI_WALKERS[i].center.y << " " << 
ORI_WALKERS[i].center.z << endl;}
	if (space_dimension == 1) {
	cout << ORI_WALKERS[i].type << " " << ORI_WALKERS[i].component << " 
" << ORI_WALKERS[i].center.x << endl; }}}
	cout << endl;
	cout.rdbuf(f);
	cout << "Code End." << endl;
	// Exit aCRD
	cout << "Exit aCRD Framework." << endl;
	releaseModule();
	system("pause");
}
```

APPENDIX D

PROJECT ACRD PROTOTYPE SOURCE CODE: SOLVER EXAMPLES

The following content provides part of the C++ source code used in Section 4. Limited by the content size, notes are not fully included in this appendix.

The code attached in this appendix is regarded to be an effective example to demonstrate the interface between solver modules and calculation system. As the source code is designed to be released with GNU Public License, the final version for aCRD system should be referred to the released code with official license.

File: MajorDiffSolver.h

```
// Major Diffusion Green's Function Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019
#ifdef MAJORDIFFSOLVER_EXPORTS
#define MAJORDIFFSOLVER_API __declspec(dllexport)
#else
#define MAJORDIFFSOLVER_API __declspec(dllimport)
#endif
struct location {
        double x;
        double y;
        double z;};

#ifdef __cplusplus
extern "C" {
#endif
        MAJORDIFFSOLVER_API double getMajorDiffTimeStamp1D(double Dv,
double domainRadius, double walkerRadius);
        MAJORDIFFSOLVER_API double getMajorDiffTimeStamp3D(double Dv,
double domainRadius, double walkerRadius);
```

```
	MAJORDIFFSOLVER_API location getMajorDiffRelativeLocation1D(double
Dv, double timestamp, double domainRadius, double walkerRadius, location
centerLocation);
	MAJORDIFFSOLVER_API location getMajorDiffRelativeLocation3D(double
Dv, double timestamp, double domainRadius, double walkerRadius, location
centerLocation);
	extern MAJORDIFFSOLVER_API double nE01SolverVersion;
#ifdef __cplusplus
	}
#endif
```

File: MajorDiffSolver.cpp

```
// Major Diffusion Green's Function Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019

#include "stdafx.h"
#include <stdlib.h>
#include <math.h>
#include <string>
#include "MajorDiffSolver.h"
using namespace std;
// Solver Version
// This is the Function for t_p, S(t_p) = (RANDOM_SEED)
// Also j(r,t_p). In this Version, use other Random Seed(s) as the Hypothesis of Even
Distribution of Diffusion
// Need Hyper-Rectangle Method in Future Version: TODO
MAJORDIFFSOLVER_API double nE01SolverVersion = 1.0;
// The Definition of Regular Parameters
#definePI 3.141592653589793238
#define eps 1e-16
double getMajorDiffTimeStamp1D(double Dv, double domainRadius, double
walkerRadius){
	double FP_TIME = 0.0;
	double TIME1 = 0.0, TIME2 = 0.0;
	double reduce_l = 0.0; // L = domainRaidus
	double reduce_t = 0.0; // t = L * L / Dv
	double RANDOM_SEED;
	double S_attempt;
	double tempt;
	double check;
```

```
    double limit1;
    double limit2;
    int flagt = 1;
    S_attempt = 0.0;
    limit1 = 0.243; // > 1.0
    limit2 = 2.000; // 5.28E-8
    reduce_l = domainRadius - walkerRadius; // Adjust unit
    reduce_t = reduce_l*reduce_l / Dv; // Adjust unit
    RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
    flagt = 1;
    while (flagt) {
        tempt = (limit1 + limit2) / 2.0;
        S_attempt = 2 * PI*PI*exp(-PI*PI*tempt); // Piecewise Smooth Function
Approach, with Long t_p Assumption
        check = abs(S_attempt - RANDOM_SEED) / RANDOM_SEED;
        if (check - 0.0000001 < eps) {flagt = 0;}
        else{
        if (S_attempt - RANDOM_SEED < eps) {limit2 = tempt;}
        else{
        limit1 = tempt;}}}
    FP_TIME = tempt*reduce_t;
    return FP_TIME;
}

double getMajorDiffTimeStamp3D(double Dv, double domainRadius, double
walkerRadius)
{
    double FP_TIME = 0.0;
    double TIME1 = 0.0, TIME2 = 0.0;
    double reduce_l = 0.0; // L = domainRaidus
    double reduce_t = 0.0; // t = L * L / Dv
    double RANDOM_SEED;
    double S_attempt;
    double tempt;
    double check;
    double limit1;
    double limit2;
    int flagt = 1;
    S_attempt = 0.0;
    limit1 = 0.243; // > 1.0
    limit2 = 2.000; // 5.28E-8
    reduce_l = domainRadius - walkerRadius; // Adjust unit
    reduce_t = reduce_l*reduce_l / Dv; // Adjust unit
```

```
        RANDOM_SEED = 5.30e-8 + rand() / (double)(RAND_MAX); // Random
SEED (0,1)
        flagt = 1;
        while (flagt) {
                tempt = (limit1 + limit2) / 2.0;
                S_attempt = 2 * PI*PI*exp(-PI*PI*tempt); // Piecewise Smooth Function
Approach, with Long t_p Assumption
                check = abs(S_attempt - RANDOM_SEED) / RANDOM_SEED;
                if (check - 0.0000001 < eps) {flagt = 0;}
                else
                {
                if (S_attempt - RANDOM_SEED < eps) {limit2 = tempt;}
                else
                {limit1 = tempt;}}}
        FP_TIME = tempt*reduce_t;
        return FP_TIME;
}

location getMajorDiffRelativeLocation1D(double Dv, double timestamp, double
domainRadius, double walkerRadius, location centerLocation)
{
        double RANDOM_SEED;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        if (RANDOM_SEED - 0.5 < eps) {
                centerLocation.x = centerLocation.x - domainRadius + walkerRadius;
        }
        else {
                centerLocation.x = centerLocation.x + domainRadius - walkerRadius;
        }
        centerLocation.x = centerLocation.x * 2.0;
        centerLocation.y = 0.0;
        centerLocation.z = 0.0;
        return centerLocation;
}

location getMajorDiffRelativeLocation3D(double Dv, double timestamp, double
domainRadius, double walkerRadius, location centerLocation)
{
        double RANDOM_SEED_THETA = 0.0;
        double RANDOM_SEED_PHI = 0.0;
        RANDOM_SEED_THETA = rand() / (double)(RAND_MAX); // Random SEED
[0,1), 2*PI*R for radians
        RANDOM_SEED_PHI = rand() / (double)(RAND_MAX); // Random SEED
[0,1), 2*PI*R for radians
```

```
        centerLocation.x = centerLocation.x + domainRadius*sin(2 *
PI*RANDOM_SEED_THETA)*cos(2 * PI*RANDOM_SEED_PHI) - walkerRadius;
        centerLocation.y = centerLocation.y + domainRadius*sin(2 *
PI*RANDOM_SEED_THETA)*sin(2 * PI*RANDOM_SEED_PHI) - walkerRadius;
        centerLocation.z = centerLocation.z + domainRadius*cos(2 *
PI*RANDOM_SEED_THETA) - walkerRadius;
        return centerLocation;
}
```

File: TransientDiffSolver.h

```
// Transient Diffusion Green's Function Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019

#ifdef TRANSIENTDIFFSOLVER_EXPORTS
#define TRANSIENTDIFFSOLVER_API __declspec(dllexport)
#else
#define TRANSIENTDIFFSOLVER_API __declspec(dllimport)
#endif

struct location {
        double x;
        double y;
        double z;
};

#ifdef __cplusplus
extern "C" {
#endif
        TRANSIENTDIFFSOLVER_API location
getTransDiffRelativeLocation1D(double Dv, double timestamp, double domainRadius,
double walkerRadius, location centerLocation);
        TRANSIENTDIFFSOLVER_API location
getTransDiffRelativeLocation3D(double Dv, double timestamp, double domainRadius,
double walkerRadius, location centerLocation);
        extern TRANSIENTDIFFSOLVER_API double nE02SolverVersion;
#ifdef __cplusplus
}
#endif
```

File: TransientDiffSolver.cpp

```
// Transient Diffusion Green's Function Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019

#include "stdafx.h"
#include <stdlib.h>
#include <math.h>
#include <string>
#include "TransientDiffSolver.h"

using namespace std;

// Solver Version
TRANSIENTDIFFSOLVER_API double nE02SolverVersion = 1.0;

// The Definition of Regular Parameters
#definePI 3.141592653589793238
#define eps 1e-16

location getTransDiffRelativeLocation1D(double Dv, double timestamp, double
domainRadius, double walkerRadius, location centerLocation)
{
        double RANDOM_SEED;
        double center_cover, random_location;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        center_cover = 2 * (domainRadius - walkerRadius);
        random_location = center_cover*RANDOM_SEED;
        centerLocation.x = centerLocation.x - domainRadius + walkerRadius +
random_location;
        return centerLocation;
}

location getTransDiffRelativeLocation3D(double Dv, double timestamp, double
domainRadius, double walkerRadius, location centerLocation)
{
        double RANDOM_SEED = 0.0;
        double RANDOM_SEED_THETA = 0.0;
        double RANDOM_SEED_PHI = 0.0;
        double center_shift;
        RANDOM_SEED = rand() / (double)(RAND_MAX); // Random SEED [0,1)
        center_shift = (domainRadius - walkerRadius)*RANDOM_SEED;
```

```
	RANDOM_SEED_THETA = rand() / (double)(RAND_MAX); // Random SEED
[0,1), 2*PI*R for radians
	RANDOM_SEED_PHI = rand() / (double)(RAND_MAX); // Random SEED
[0,1), 2*PI*R for radians
	centerLocation.x = centerLocation.x + center_shift*sin(2 *
PI*RANDOM_SEED_THETA)*cos(2 * PI*RANDOM_SEED_PHI);
	centerLocation.y = centerLocation.y * center_shift*sin(2 *
PI*RANDOM_SEED_THETA)*sin(2 * PI*RANDOM_SEED_PHI);
	centerLocation.z = centerLocation.z * center_shift*cos(2 *
PI*RANDOM_SEED_THETA);
	return centerLocation;
}
```

File: LocalSolver.h

```
// Local Reaction Possibility Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019

#ifdef LOCALSOLVER_EXPORTS
#define LOCALSOLVER_API __declspec(dllexport)
#else
#define LOCALSOLVER_API __declspec(dllimport)
#endif

struct location {
	double x;
	double y;
	double z;
};

// This structure is designed for SPD_a
struct domaincenter {
	double x;
	double y;
	double z;
	double r;
};

// The walkers follow domain time (protected) or global time (released)
struct walker {
    long ID;
    int type; // 0 V_Based; 1 I_Based; -1 for Not Using
            // int subtype; // For different defect structures, not in use for
current version.
    long component; // Cluster number; 1 for single walker
    location center;
    double radius; // Use modifier for vacancies
    double energy; // In keV
    double diffusivity; // In nm^2/s
    int statue; // 0 Released; 1 Soft Protected; 2 Hard Protected
    long regionID;
    double timestamp; // WORLD
};

// For current version, only pre-assigned final status: 1 - 50 walkers
struct reaction_current_walker {
    walker member[50];
};

// For current version, only pre-assigned final status: 1 - 5 walkers
struct reaction_final_walker {
    walker member[6];
};

#ifdef __cplusplus
extern "C" {
#endif
    LOCALSOLVER_API double
getReactionTimeStamp1D(reaction_current_walker walkers, double initial_time);
    LOCALSOLVER_API double
getReactionTimeStamp3D(reaction_current_walker walkers, double initial_time);
    LOCALSOLVER_API reaction_final_walker
getReactionFinalStatues1D(reaction_current_walker walkers, int init_member,
domaincenter centerA, domaincenter centerB);
    LOCALSOLVER_API reaction_final_walker
getReactionFinalStatues3D(reaction_current_walker walkers, int init_member,
domaincenter centerA, domaincenter centerB);
    extern LOCALSOLVER_API double nE10SolverVersion;
#ifdef __cplusplus
}
#endif
```

File: LocalSolver.cpp

```
// Local Reaction Possibility Solver
// In MOD_SOL_aCRD
// Version 1 alpha demo
// J. Fan 2018-2019

#include "stdafx.h"
#include <stdlib.h>
#include <math.h>
#include <string>
#include "LocalSolver.h"

using namespace std;

// Solver Version
// This solver is using t_built+FIX_EVENTS_TIME*(1+RANDOM_SEED) as t_event
// This solver only concerns coalescence, with 1 final walker in random center
// This solver doesn't provide SPDa time update interface, only member update in main()
LOCALSOLVER_API double nE10SolverVersion = 1.0;

// The Definition of Regular Parameters
#definePI 3.1415926535897932 38
#define eps 1e-16
// This is only for first initialization estimation
// #define INITIAL_EVENTS_TIME 0.001

double getReactionTimeStamp1D(reaction_current_walker walkers, double initial_time)
{
        double RX_TIME = 0.0;
        double RANDOM_SEED;
        // SPDa will trigger within INITIAL_EVENTS_TIME +
INITIAL_EVENTS_TIME * Random[0,1)
        RANDOM_SEED = rand() / (double)(RAND_MAX);
        RX_TIME = initial_time + initial_time*RANDOM_SEED;
        return RX_TIME;
}

double getReactionTimeStamp3D(reaction_current_walker walkers, double initial_time)
{
        double RX_TIME = 0.0;
        double RANDOM_SEED;
        // SPDa will trigger within INITIAL_EVENTS_TIME +
INITIAL_EVENTS_TIME * Random[0,1)
```

```
        RANDOM_SEED = rand() / (double)(RAND_MAX);
        RX_TIME = initial_time + initial_time*RANDOM_SEED;
        return RX_TIME;
}

// Annihilation only for this version
reaction_final_walker getReactionFinalStatues1D(reaction_current_walker walkers, int
init_member, domaincenter centerA, domaincenter centerB) {
        reaction_final_walker final_walkers;
        final_walkers.member[1].type = -1;
        final_walkers.member[2].type = -1;
        final_walkers.member[3].type = -1;
        final_walkers.member[4].type = -1;
        final_walkers.member[5].type = -1;
        return final_walkers;
}

// Coalescence to 1 or Annihilation in this version
// Need walker type, component, center
reaction_final_walker getReactionFinalStatues3D(reaction_current_walker walkers, int
init_member, domaincenter centerA, domaincenter centerB) {
        reaction_final_walker final_walkers;
        int i_counter;
        int v_counter;
        long i_all_comp;
        long v_all_comp;
        int i;
        double RANDOM_SEED = 0.0;
        i_counter = 0;
        v_counter = 0;
        i_all_comp = 0;
        v_all_comp = 0;
        for (i = 0;i < init_member;i++) {
                if (walkers.member[i + 1].type == 0) {
                        v_counter = v_counter + 1;
                        v_all_comp = v_all_comp + walkers.member[i + 1].component;
                }
                if (walkers.member[i + 1].type == 1) {
                        i_counter = i_counter + 1;
                        i_all_comp = i_all_comp + walkers.member[i + 1].component;
                }
        }
        if (i_all_comp == v_all_comp) {
                // Annihilation
        final_walkers.member[1].type = -1;
        final_walkers.member[2].type = -1;
        final_walkers.member[3].type = -1;
        final_walkers.member[4].type = -1;
        final_walkers.member[5].type = -1;
    }
    else {
        if (i_all_comp < v_all_comp) {
            // Coalescence to vacancy cluster
            final_walkers.member[1].type = 0;
            final_walkers.member[1].component = v_all_comp - i_all_comp;
            // Select one of SPDa center as final center, try best not to break
other HPD
            RANDOM_SEED = rand() / (double)(RAND_MAX); // Random
SEED [0,1)
            if (RANDOM_SEED - 0.5 < eps) {
                final_walkers.member[1].center.x = centerA.x;
                final_walkers.member[1].center.y = centerA.y;
                final_walkers.member[1].center.z = centerA.z;
            }
            else {
                final_walkers.member[1].center.x = centerB.x;
                final_walkers.member[1].center.y = centerB.y;
                final_walkers.member[1].center.z = centerB.z;
            }
            final_walkers.member[2].type = -1;
            final_walkers.member[3].type = -1;
            final_walkers.member[4].type = -1;
            final_walkers.member[5].type = -1;
        }
        else {
            // Coalescence to interstitial cluster
            final_walkers.member[1].type = 1;
            final_walkers.member[1].component = i_all_comp - v_all_comp;
            // Select one of SPDa center as final center, try best not to break
other HPD
            RANDOM_SEED = rand() / (double)(RAND_MAX); // Random
SEED [0,1)
            if (RANDOM_SEED - 0.5 < eps) {
                final_walkers.member[1].center.x = centerA.x;
                final_walkers.member[1].center.y = centerA.y;
                final_walkers.member[1].center.z = centerA.z;
            }
            else {
                final_walkers.member[1].center.x = centerB.x;
                final_walkers.member[1].center.y = centerB.y;
                final_walkers.member[1].center.z = centerB.z;
            }
            final_walkers.member[2].type = -1;
            final_walkers.member[3].type = -1;
            final_walkers.member[4].type = -1;
            final_walkers.member[5].type = -1;
        }
    }
    return final_walkers;
}
```