International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:2, 2012

A Method to Annotate Programs with High-Level Knowledge of Computation

Nobuhiko Hishinuma, Jun Igari, and Rentaro Yoshioka

***Abstract*—**When programming in languages such as C, Java, etc., it is difficult to reconstruct the programmer's ideas only from the program code. This occurs mainly because, much of the programmer's ideas behind the implementation are not recorded in the code during implementation. For example, physical aspects of computation such as spatial structures, activities, and meaning of variables are not required as instructions to the computer and are often excluded. This makes the future reconstruction of the original ideas difficult. AIDA, which is a multimedia programming language based on the cyberFilm model, can solve these problems allowing to describe ideas behind programs using advanced annotation methods as a natural extension to programming. In this paper, a development environment that implements the AIDA language is presented with a focus on the annotation methods. In particular, an actual scientific numerical computation code is created and the effects of the annotation methods are analyzed.

***Keywords*—**cyberFilm, development environment, knowledge engineering, multimedia programming language

I. Introduction

IN traditional text-based programming languages such as C, Java and the like, computation is described by only a sequence of commands. The objective of these commands are to command the computer, not to explain about the program to programmers.

When programmers try to understand the program, they often rely on document. Therefore, programmers should create and maintain high-quality documents. To maintain its quality, suitable modification of the document is required according to change of the program. But in reality, this activity is not performed appropriately [1]. As a result, programmers must eventually understand the program by reading its code [2]. One of the ways of making programs readable and understandable is by using identifier names and comments [3]. But, if these names and comments are not appropriate, the programmers will be confused easily. Furthermore, it is often useful to understand and visualize the structural construction of the environment related to the actual phenomenon. Without this knowledge, not only much labor is spent, but also misunderstanding occurs. To explain such feature, identifier names and comments should be written in detail and understandable.

Nobuhiko Hishinuma is with the Computer Science and Engineering Department, University of Aizu (e-mail: m5141201@u-aizu.ac.jp).

Jun Igari is with the Computer Science and Engineering Department, University of Aizu (e-mail: m5141202@u-aizu.ac.jp).

Rentaro Yoshioka is with the Computer Science and Engineering Department, University of Aizu (e-mail: rentaro@u-aizu.ac.jp).

But if structure of computation is spatial and complex, it is so difficult to describe the structure by only text. Images can be useful in describing such features, but the reliability of document is low.

Even if a programmer has sufficient background knowledge of the computation, still it is a hard task to map the line of code of its implementation with one's understanding. This difficulty occurs since he spatial and temporal aspects of physical phenomena are not coded into the instructions of a programming language. Example of such aspects includes structure, flow of computation and meaning of variables. Suppose, for example, there is computation about particle collision. Some programmers know the particles perform either fission, scatter, or capture upon collision. But, if there are no such direct explanations in words in the implemented program, the programmers need to analyze the program line by line. It is difficult to understand the programmer's ideas only from implemented code [1]. Therefore, as much knowledge related to the environment, objective of the computation, and the implementation strategies should be recorded along with the program.Even if a programmer knows the computation well, the programmer can be easily confused by unrelated codes. In many cases, there are such codes in a program just to make it work. For example, input operations are not so important to understand the computation itself. But, these codes often take up rather great part of programs. Therefore, programmers spend energy to distinguish main computation from supportive codes. To focus on only main computation, a mark to distinguish them will be required.In the past, various programming approaches such as object-, aspect-, component- oriented programming have been provided. But, they are methods to just coordinate program or reduce waste. To address these problems, not only that, but the program should be able to record high-level knowledge. But traditional programming languages and methods based on them cannot have the knowledge in the program, because their specification is just to write sequential commands in only text.To solve these problems, the Animations and Images for Development of Algorithms (AIDA) language and Active Knowledge Studio (AKS) have been developed. The AIDA language is a multimedia programming language based on cyberFilm method. The cyberFilm method is a format to represents computation using multimedia components (icons, animations and extended-texts) [6]–[8]. A computation usually has various features such as structure, flow, data, and interface. The AIDA language consists of four different languages and they represent these features: the Language of Algorithmic

Dynamics (LAD), the Language of Algorithmic Commands (LAC), the Language of Algorithmic Interface (LAF) and the Language of Algorithmic Text (LAT). The LAD represents the structure and flow of computation. The LAC represents the activity of computation by variables and formulas. The LAF represents the operation related to input/output data. The LAT represents all the previously described features all together in a condensed form. AKS is a development environment to implement program in the AIDA language, and provides five views to edit and browse programs in the four languages and execute the program: the Skeleton View for LAD, the Formula View for LAC, the IO View for LAF, the Integrated View for LAT and the Run View to transform the AIDA language program into other languages such as C, Java, FORTRAN and execute the program.The AIDA language can not only implement the computation like other languages, but also allows to attach annotations about the programmer's ideas. AKS supports this effectively by taking advantage of multiple views. The AIDA language and AKS is not only to computation, but also to model and document them. Using the AIDA language on AKS, programmers can extract programmer's ideas from implemented programs in such a way as browsing documents. In this paper, the AIDA language is applied to an example computation which is to solve the Boltzmann equation by the Monte Carlo method [4], [5], and comparing it with FORTRAN (a traditional text-based programming language). Through this comparison, the effectiveness in understanding computation ideas of the AIDA language and AKS are analyzed and represented.This paper consists of follows. In section 2, overview of the target computation is explained. In section 3, the Integrated View and its effectiveness is explained by representing examples applied to the example computation and comparing with FORTRAN program. In section 4, the functions of other views (Skeleton, Formula, IO and Run view) are explained, and the availability of AKS as a development environment is represented. Conclusion and future work are shown in section 5.

II. Target Computation

A. Over View

The example computation we consider is the computation to solve the Boltzmann equation by the Monte Carlo method. The purpose of this computation is to obtain effective increase of neutrons by solving Boltzmann equation with Monte Carlo approach. In addition, some control data and statistics data are also computed in this computation. In this section, the computation is explained with the flow shown in Fig. 1.

This computation says the Boltzmann equation, but the expression of the Boltzmann equation is not appeared. The Boltzmann equation is known as integrodifferential equation with considering elementary steps such as streaming, collision, fission, scatter, and capture of neutron particles. But, in the computation to solve by Monte Carlo method, the equation is not required because each particle are traced and calculated stochastically.

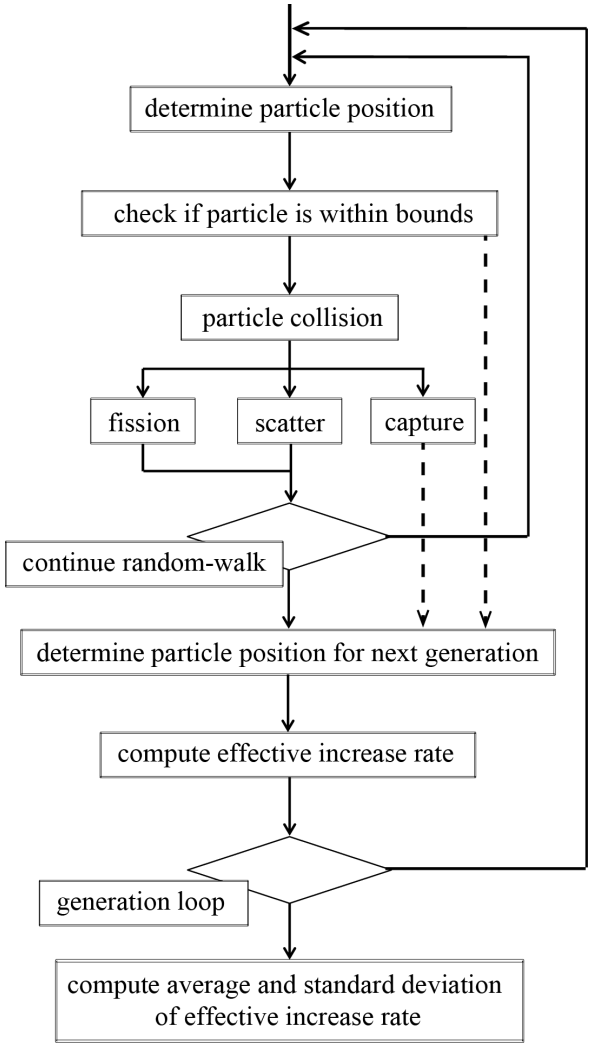


Fig. 1 Flow of Computation

This computation simulates particles transportation in space with range of x-axis (the length of y-axis and z-axis is infinity). The particles move and collide in this space randomly. First, the particle position after random-walking is determined, and the position is checked if particle is within the range of this space. If the particle is not in the range, the particle is judged as leaked particle, and the random-walking is terminated. If the particle is in the range, the particle collision is computed.

The colliding particle performs a reaction from three types of reaction:

1) *Fission:* the colliding particle splits into some particles and changes the moving direction.
2) *Scatter:* the colliding particle changes the moving direction.
3) *Capture:* the colliding particle is captured by other particle.

If the particle is captured, the random-walking is terminated. If fission or scatter is selected, the random-walking is continued until the particle goes to out of range or captured. After the random-walking, particles position which are neutrons emitted in this generation are determined for next generation, and

effective increase rate is recorded. Then, the loop of the generation is carried out the number of times which is decided before computation. Finally, average and standard deviation of effective increase rate are computed.

B. Implementation Method

To implement this computation in existing programming language such as FORTRAN, some information are often added or transformed. Through the process, some problems to understand computation are raised. In this section, these problems are represented and explained.

1. Structure of Computation

The main structure of this computation is spatial structure which represents moving particles in space with range of x-axis. Then, almost all flow and activities of the computation shown in previous section is done based on this structure. In the FORTRAN program, the information of this structure is separated into some parts such as variables and formulas of computation. Fig. 2 represents transformation of the structure to FORTRAN specification.

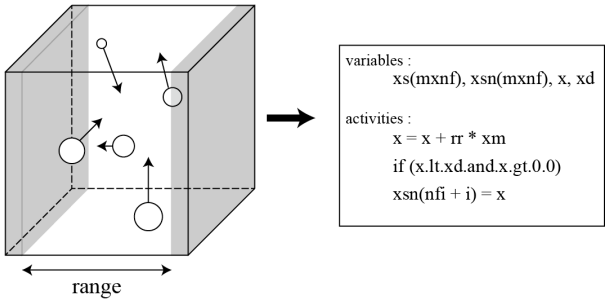


Fig. 2 Structure in FORTRAN's Implementation

The structure which is initially intended by the programmer is shown in the left of Fig. 2 as an image. This image represents the structure of moving particles in space with range of x-axis. But to actualize this structure by the FORTRAN language, programmers will write program codes such as the right of Fig. 2. In this example, the variables *xs(mxnf)* and *xsn(mxnf)* store the x-position of the *mxnf* number of particles, and *x* which also store x-position of particle is for calculation. The variable *xd* is for x-width of range. In the activities, first expression represents random-move of the particle (*rr* and *xm* decided by random number). The condition which is second command in the activity represents that this structure has range. The final activity represents the relationship between *xsn(mxnf)* and *x* (i.e. *xsn(mxnf)* is to store position for next generation and *x* is for calculation).

These transformed parts of information are spread in the FORTRAN program but they are intimately related to each other. Therefore, programmers will spend much time and energy trying to read the program with up and down to understand the structure. Moreover, to understand the some activities such as these particles have collision in this space, programmers will spend more cost to read. The structure of computation is very important factor to understand program and objectives of computation, so various programmers try to understand it before understanding computation. But, these difficulty and complexity to read program and understand the structure increase the cost of understanding computation and sometimes raises misunderstanding.

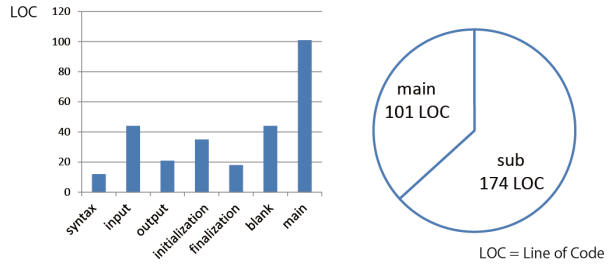
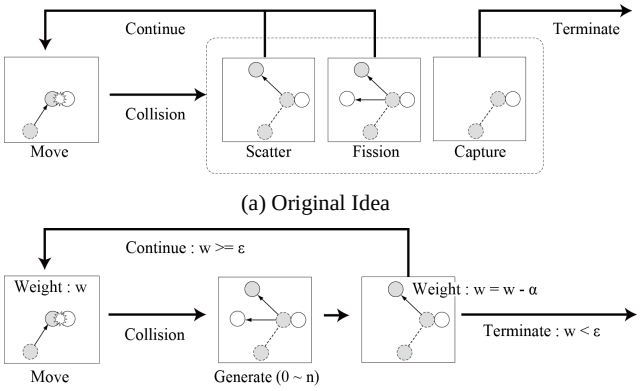


Fig. 3 Number of Lines par Part

2. Sub Computation for Application

A computation often includes computations which are unrelated to its objective directly such as input, output, initialization, and finalization. In many cases, such supportive computations for main computations are not needed to understand the computation. But, such sub computation often takes up various part of program and prohibits programmers from reading codes. Fig. 3 represents number of lines par part in the FORTRAN program applied to the example computation. In this program, over 60 percent of program is used for sub part and they are intermixed. These unnecessary computations to understand often confuse programmers when they read and modify the program.



(a) Original Idea

(b) Implementation Method

Fig. 4 Flow of Particle Collision

3. Optimization for Computation

The one of the most important points of the example computation is to determine type of reaction when particles are collided. If the program is implemented according to this original idea devotedly, it may have conditional branching to compute a reaction from three types of reaction randomly and reiterate it such as Fig.4 (a). But, there are some cases that the implementation disobedient to the flow of original ideas for optimization. The FORTRAN program is also disobedient to

this flow about particle collision. In this program, particle collision is computed by follow steps (Fig. 4 (b) shows this flow):

1) recode number of new particles and each position (zero to n new particles are generated randomly)
2) reduce weight of the particle
3) terminate the particle if it has low weight

Using features of neutron, this computation has succeeded to reduce unless part for implementation and quicken processing speed of simulation. But, at the same time, this computation lost information about the original idea as particle collision and three types of reaction. As a result, programmers will have difficulty to find where the computation is done in the FORTRAN program. It becomes difficult to understand the original idea of computations from the FORTRAN program.

III. Annotation Method for High-Level Knowledge

To describe programmer's ideas before implementation, the AIDA language have some annotation methods. In this section, the annotation methods to describe programmer's ideas are analyzed and represented through comparing the Integrated View with FORTRAN.The Integrated View, which is one of the views for the AIDA language, can represent all the features of computation in a condensed form, and can edit and browse the computation optimally. For example, the Integrated View applied to the example computation is shown in Fig. 4. In the Integrated View, all computations are represented by icon and extended text which can represent some particular text such as mathematical symbols. This Integrated View needs only one-and-a-half A4 paper compared with that the FORTRAN program needs four A4 papers. The Integrated View consists of header and body section.

1. Header Section

Header Section is the top rectangular of Integrated View and represents all structures and variables of computation. Structures are represented by structure name, structure icon and parameters of structure. The type of structure (e.g. 2D-Grid, 3D-Grid and moving particles) is determined by structure icon. If there are same types of structure in the Integrated View, they are identified by parameters and name of structure. On the other hand, variables are declared by structure icon, format icon and name. Structure icon of variable represents shape of variable such as scalar, 1-, 2-, 3-D grid, moving particles. Format icon of variable represents the type of variable such as integer, float, double float, string. In addition, variables can also have information of unit and group by icon [11]. For example, the variable *xd* of Fig. 2 can have Nanometer (nm) as unit and *width* as group using icon.

2. Body Section

Body Section is under the header section and represents flows and activities of computation. The body section has some hierarchical sections called scene which are structuralized by parts of computation. Each scene is represented by scene icon and terminal section which consists of node icons and formulas.

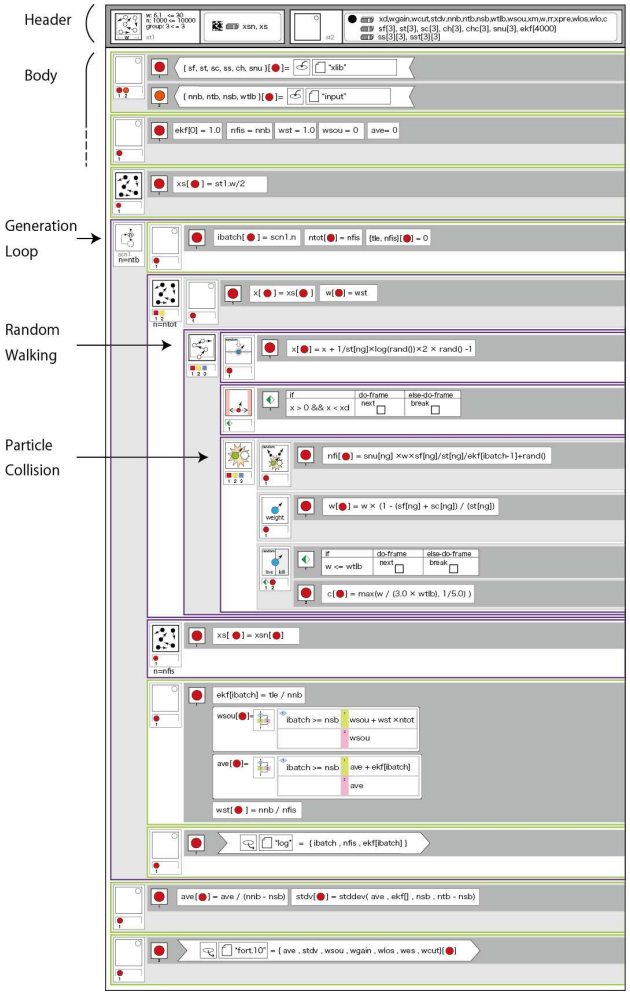


Fig. 5 Integrated View or The Example Computation

Flow of computation is represented by scene icon and node in header. For example, the scene Random Walking and Particle Collision in Fig. 5 are flow for the structure *moving particles* and the node icon represents each particle in the structure. But the temporal scene likes Generation Loop (While Loop) in Fig. 5 represents structure by themselves.

A. Structure of Computation

Even if a programmer has sufficient knowledge of the computation, it is difficult to relate an implemented code to physical phenomenon such as spatial structures and activities. The reason is that such spatial and temporal information is fragmented and embedded in code with other information.

To solve this problem, the AIDA language provide environment to describe such structural construction without fragmentation. For example, programmers can select and determine structures of computation intuitively compared with traditional programming languages such as FORTRAN using the Integrated View. Fig. 6 is example to represent the structures and variables of the example computation. In this example, the information related to the structure, which is moving particles in space with range, is shown in the upper right in Fig. 6. This

structure has three parameters, *w*, *n* and *group*. *w* means the range of space, *n* means the number of particles, and *group* means the number of types of particle. *xsn* and *xs* are variables based on this structure to store the position of particles. On the other hand, the other section of Fig. 6 shows observer structure which is related to others such as computations of statistics and observation.

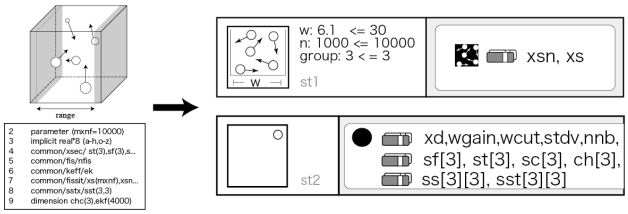


Fig. 6 Example of Header

Compared with the declaration of structures in the FORTRAN program (Fig. 2), the information of the structures is gathered in this section. Therefore, this approach enables users to understand what types of structure are computed in the program before reading the body of code. In addition, users can image and understand spatial structures intuitively by graphical icons even if they are not specialist of the example computation. This approach will reduce the cost and misunderstanding in the process of preparation to understand main computation.

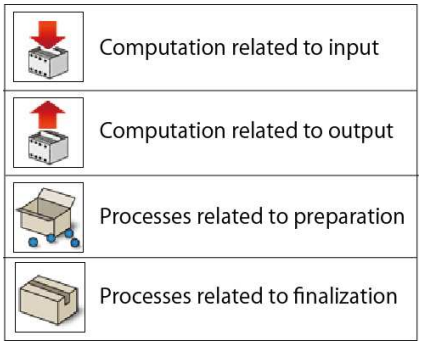


Fig. 7 Types of Sub Computation

B. Sub Computation for Application

As the structure in the Integrated View, flows of computation can be understood by scene icons before starting to read the contents of scenes. Activities of computation such as formulas will be also more easily to understand than FORTRAN, because the Integrated View can use mathematical symbols such as (e.g. Σ, π and χ_ρ).

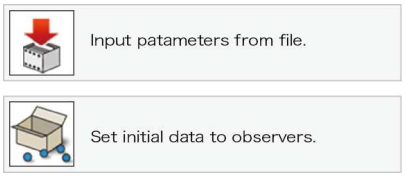
The Integrated View can represent not only implementation easily, but also can represent explanation of computations effectively. Scenes can have icons and comments as annotations to explain the computation. Then, users can display the annotations by folding scenes for implementation. Using this function, users can obtain two types of effectiveness.

The one is that users can read program with obtaining the part of main computation. Parts of computation can be classified into main computation and sub computation. The main computation means important part to understand computations. On the other hand, the sub computation means not important part to understand it such as input, output, initialization, and finalization. The main computation is based on the example computation, so there is just as various types for that as there are computations. But, we can anticipate types of the sub computations to some extent. For example, some types of sub computation are represented in Fig. 7 with icons. These icons for sub computation are predefined and programmers can apply unified it to a program. Therefore, programmers can assess the code is important or not easily.

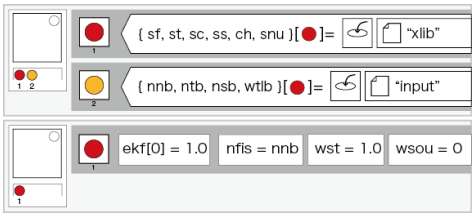
Fig. 8 represents example to show the different between the FORTRAN program, unfolded scenes, and folded scenes. The top section of Fig. 8 (b) and Fig. 8 (c) represent activity of input operations for variables of observer structure, and another section represents activity of initialization for variables of observer structure.

```
12    read(7,*)
 :       :
27    read(7,*)snu(1),snu(2),snu(3)
28    c
29    chc(1)=ch(1)
 :       :
35    sst(nn,2)=(ss(nn,1)+ss(nn,2)...
36    end do
```

(a) FORTRAN



(b) Folded



(c) Unfolded

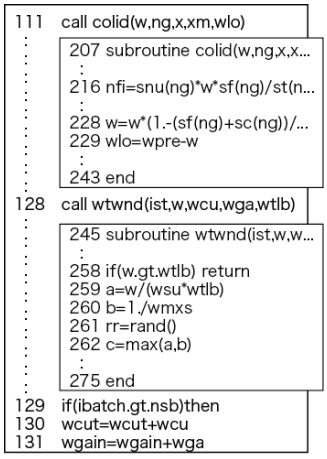
Fig. 8 Sub Computation Scenes and Its Explanation Scenes

Compared with the FORTRAN program such as Fig. 8 (a), the computations are distinguishable on the basis of prepared icons to explanation. Therefore, users can not only avoid reading meaningless computation, but also minimize size of program. Additionally, whenever users want to read or change such sub computation scene, users can find the point with the explanation. This approach will reduce the cost for searching main computation from large program.

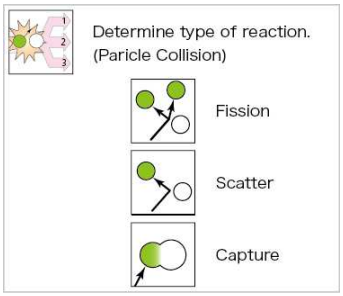
About the other effectiveness of annotations is represented in next sub section.

C. Optimization for Computation

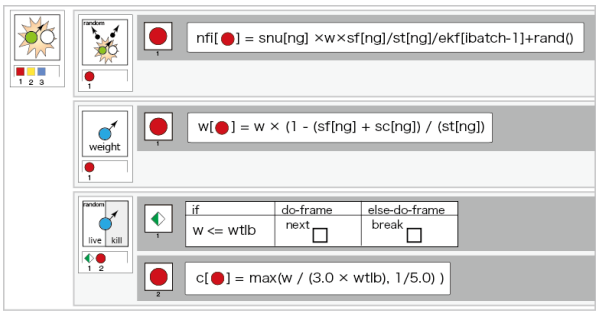
The other effectiveness to use the annotations for scenes is that users can read and understand both implementation method and original ideas which are initially intended by programmers.



(a) FORTRAN



(b) Folded



(c) Unfolded

Fig. 9 Scenes for Implementation Method and Original Ideas

Fig. 9 also represents flow and activity of the example computation, but this scene is compute main computation for particle collision. This unfolded scene is implemented by implementation methods such as the FORTRAN program (Fig. 9 (a)). Therefore, there are three scenes to explain computation to calculate number of particles genesis, computation weight of particle and termination particle according to its weight in Fig.9 (c). This scene also has annotations which are represented by Fig. 9 (b). Whereas the scenes of Fig. 9 (c) explain the behavior of computation, this annotation represents information based on programmer's original ideas; particle collision and three types of reaction. As you can see, the Integrated View can assign and represent computation based on implementation methods as well as computation according to the ideas. As a result, users can understand the rationale behind computation instead of documents. This function can record the programmer's knowledge and purpose, and propose it to readers without misunderstanding. This approach will reduce the risk of misunderstanding the computation and the cost to understand the computation.

In addition, the deference from the FORTRAN program is that not only the computation is represented by visual components, but also some formulas which are not necessarily to understand or modify computation are hided. For example, Fig. 10 represents the FORTRAN program and a scene of the Integrated View about computation to calculate number of particles genesis. In FORTRAN program, *nfi* which means number of particles genesis is computed at first. Then, *xsn* which means the positions of generated particles is recorded (L220 to L222) and *nfis* which means total number of particles is uploaded (L224). But, only the computation to calculate number of particles genesis is appeared in the Integrated View and the others are hided. Because, even if the way of the computation for number of particles genesis is changed, the computation for recording positions and uploading total number will be needed. As a result, the Integrated View becomes more compactness than traditional programming languages and provides strong information encapsulation.

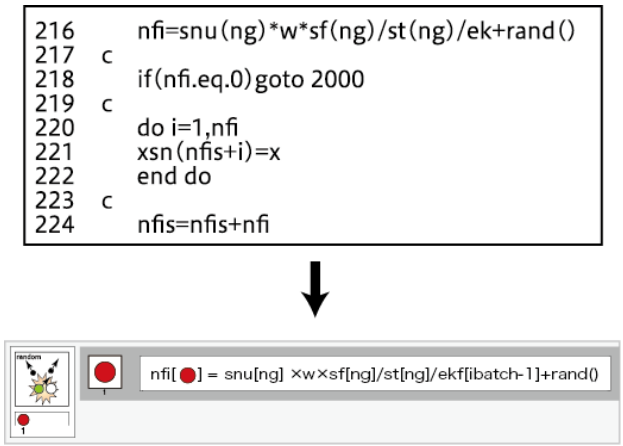


Fig. 10 Extraction of Reconstructive Computation

IV. Other Method for Annotation

Besides the Integrated View, AKS have four views to understand features of computation and to control the program more easily. Fig. 11 represents relationship between views of AKS. These views are synchronized with together, and they can support to watch and edit the implementation. The Skeleton View, the Formula View and the IO View represent features of

computation along with the Integrated View. The deference of these views and the Integrated View, the Integrated View represents whole features of combination but these views represent each specialized features. The Run View does not represent features of combination supported by the AIDA language but it is also important view for AKS. After implementation in the AIDA language, it is executed in the Run View. In this section, description of each view is represented.

1. Skeleton View

The Skeleton View focuses on structures and flows of computation according to LAD. Detailed information of flow of the computation can be watched by animations and images as annotations. For example, Fig. 12 is image of random-walking scene which can be watched on the Skeleton View. If only the Integrated View, users may be able to understand only the overview of structure, but users can understand even the more detailed action of structure using this view. Additionally, users can also edit the information in this view.

2. Formula View

The Formula View focuses on formulas and activities of computation according to LAC. When users want to edit formulas, this view will be often used. Of cause, basic formula can be edited in the Integrated View, but there are some particular formulas such as structured expression in specification of the LAC.

For example, Fig. 13 represents two types of structured formula. Fig. 13 (a) represents computation for conditional branching. This formula means that the expression $wsou = wsou + wst * ntot$ is computed when $batch = nsb$ is true. Fig. 13 (b) represents computation for long or complex expression and means (1). Using these particular formulas, programmers can describe expressions more briefly without temporal variables. The Formula View is prepared to edit such expression easily because the view is specialized to edit and watch formulas.

$$G = (\alpha w + \frac{\varepsilon K \sigma}{tempr - \theta a}) \quad (1)$$

3. IO View

The IO View focuses on input and output between the AIDA language and any components according to LAF. In this view, users can select, edit and check input/output files. If input/output files are selected on the Integrated View, the information is reflected to this view.

4. Run View

There is the Run View to build and execute the program written in the AIDA language with AKS. This view generates program code in other programming language by template programming [9], [10]. After the program generation, the AIDA language becomes compile-able program as other languages. Then, the program can be executed on this view directly. The IO data can be confirmed on the IO View.

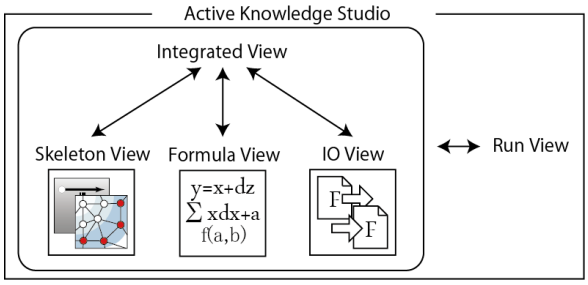


Fig. 11 Relationship of Views

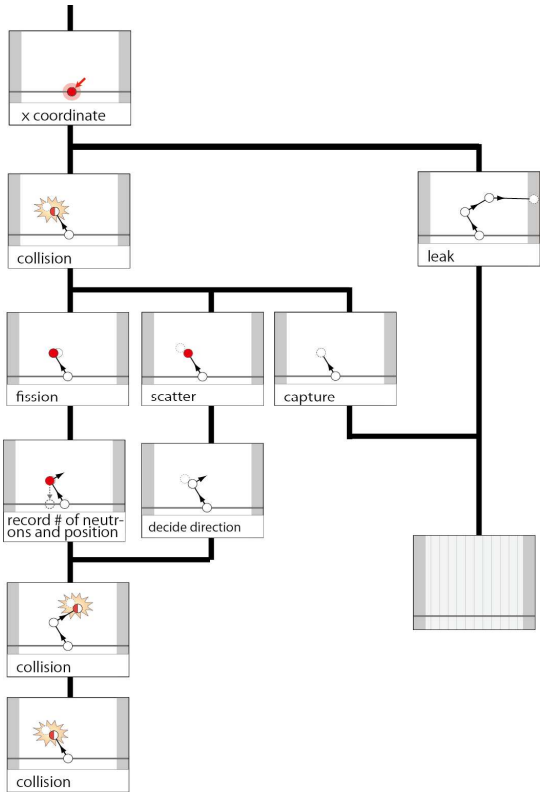
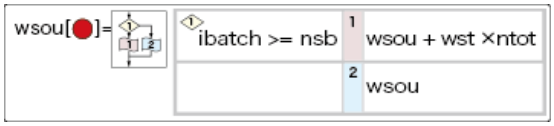
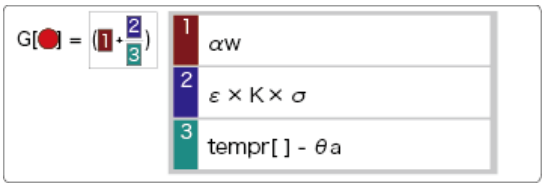


Fig. 12 Image the Flow of Computation



(a) Inline IF Formula



(b) Inline Pattern Formula

Fig. 13 Structured Formulas

V. Conclusion and Future Works

In summary, we have proposed three types of annotation methods to record programmer's ideas with high-level knowledge. The first annotation method is to describe the structural construction of the application environment definition and icons. As a result, even if programmers do not have prior knowledge, they can understand the structure easily without analyzing program code. The second annotation method is to readily distinguish between main computation and supportive computation by marking them a predefined classification of tags. Using this annotation method, programmers can focus on main computation to analyze easily. The last annotation method is to explain high-level knowledge such as objectives of computation and implementation strategies. Through understanding of the knowledge, programmers can understand the programmer's original ideas and its relationship with implementation methods. Previously, a programmer's efficiency and quality of understanding a program depends mostly on individual ability. But, these annotation methods enable programmers who develop the program to suggest the way of understanding the computation. This approach reduces not only labor of understanding, but also the risk of misunderstanding.

In addition, a development environment called AKS for the AIDA language has been implemented to demonstrate these methods, and the computation to solve the Boltzmann equation by the Monte Carlo method was modeled and implemented. As an evaluation, these methods and applications have obtained a good reputation from the developers using the example computation. Additionally, through the development of AKS, various program specification and visualization techniques in each view were developed.

As future work, the development of AKS is continuing along with the improvement of the AIDA language. In particular, more information to understand programmer's ideas such as about variables, formulas and input/output contents will be implemented. Other functions, such as searching of annotations, debugging a program at the level of annotations are also considered.

Acknowledgment

The sample computation and related FORTRAN program presented in this paper were provided from Japan Nuclear Energy Safety Organization (JNES). We appreciate JNES for their cooperation. Also we thank our laboratory members for their support in developing AKS.

References

[1] T. D. LaToza, G. Venolia and R. DeLine, “Maintaining Mental Models: A Study of Developer Work Habits”, ICSE, New York, 2006.
[2] T. D. LaToza, D. Garlan, J. D. Herbsleb and B. A. Myers, “Program Comprehension as Fact Finding”, ESEC-FSE, New York, 2007.
[3] D. Lawrie, C. Morrell, H. Feild and D. Binkley, “What's in a Name? A Study of Identifiers” In 14th International Conference on Program Comprehension.
[4] S. A. Dupree, S. K. Fraley, “A Monte Carlo Primer: A Practical Approach to Radiation Transport”, Kluwer Academic/Plenum Publisher, New York, 2002.
[5] S. A. Dupree, S. K. Fraley, “A Monte Carlo Primer Volume 2: A Practical Approach to Radiation Transport”, Kluwer Academic/Plenum Publisher, New York, 2004.
[6] N. Mirenkov, A. Vazhenin, R. Yoshioka, T. Ebihara, T. Hitomi and T. Mirenkova “Self-Explanatory Components: a New Programming Paradigm”, International Journal of Software Engineering and Knowledge Engineering, 11(1), 5-36, 2001.
[7] N. Mirenkov and R. Yoshioka, “Visual Computing Within Environment of Self-explanatory Components”, Soft Computing Journal 7, 20-32, 2002.
[8] N. Mirenkov and R. Yoshioka, “A Multimedia System to Render and Edit Self-Explanatory Components”, The Journal of Internet Technologies, 3(1), 1-10, 2002.
[9] Y. Watanobe, N. Mirenkov and R. Yoshioka, “Algorithmic CyberFilm Language”, FCST '06, Japan-China Joint Workshop, 2006.
[10] T. Ebihara, “A Program Generator from CyberFilm Specifications”, unpublished, University of Aizu, 2005.
[11] K. Takeshige, “A Language of Embedded Clarity Support”, unpublished, University of Aizu, 2011.