A Formal Software Development Approach Using Refinement Calculus

WANG Yunfeng (王云峰)[1,2], PANG Jun (庞 军)[1],
ZHA Ming (查 鸣)[1], YANG Zhaohui (杨朝晖)[1] and ZHENG Guoliang (郑国梁)[1]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University Nanjing 210093, P.R. China*

[2] *Meteorology College, PLA University of Science and Technology, Nanjing 211101, P.R. China*

E-mail: zhenggl@nju.edu.cn

Received July 26, 1999; revised April 14, 2000.

Abstract The advantage of COOZ (Complete Object-Oriented Z) is to specify large scale software, but it does not support refinement calculus. Thus its application is confined for software development. Including refinement calculus into COOZ overcomes its disadvantage during design and implementation. The separation between the design and implementation for structure and notation is removed as well. Then the software can be developed smoothly in the same frame. The combination of COOZ and refinement calculus can build object-oriented frame, in which the specification in COOZ is refined stepwise to code by calculus. In this paper, the development model is established, which is based on COOZ and refinement calculus. Data refinement is harder to deal with in a refinement tool than ordinary algorithmic refinement, since data refinement usually has to be done on a large program component at once. As to the implementation technology of refinement calculus, the data refinement calculator is constructed and an approach for data refinement which is based on data refinement calculus and program window inference is offered.

Keywords formal development method, refinement calculus, formal specification, object-oriented

1 Introduction

The confidence in a program's correctness can be obtained by describing its intended task in a formal notation. Such specification can then be used as a basis for a provably corrected development of the program. The development can be conducted in small steps, thus allowing the unavoidable complexity of the final program to be introduced in manageable pieces.

The process, called refinement, by which specifications are transformed into program has been extensively studied. (see [1] for an overview of the current work). In particular [2, 3] have laid down much of the theory and have recognized two forms of refinement. The first is algorithmic refinement, by which a program operates more explicitly, usually introducing an algorithm to replace the statement of desired result. The second is data refinement, where one changes the structure for storing information, usually replacing some abstract structure that is easily understood by some more concrete and efficient structure.

The extension from Dijkstra's language to the refinement calculus was made by Back[4], then redeveloped independently by Morris[5], Morgan[6] and by Back and von Wright[7].

Gardiner and Morgan[8] argued a style of data refinements which are calculated directly without proof obligation. It is advantageous over the methods of Spivey[9] and Stepenny *et al.*[10], which need obligation to prove data refinement.

The work is supported by the National Natural Science Foundation of China (No.69673006) and the National Ninth Five-Year Project (98-780-01-07-06) of China.

Previous attempts to address formal development fall into two categories. In the first one, a specialized calculus is developed within Z or COOZ (Complete Object-Oriented Z) to allow algorithm refinement. An example of this approach can be found in [1]. The second approach involves a translation stage in which the Z specification is transformed into another notation, which is more amenable to refinement. For example, King[12] gave the rules for translating Z specification into Morgan's refinement calculus. Then the refinement calculus law can be used to develop the specification to code.

Both of above approaches seem somewhat wasteful. On one hand, because the translation stage needs much effort but not contributes directly to development, and on the other hand, because so much work has gone into the development of various flavors of refinement calculus which already exist. In particular, Morgan's[13] calculus is well developed and its law has been codified and collected in such a way, which can be used to develop real programs from abstract specification.

So, for the refinement of object-oriented specification, we hope to develop an approach which integrates the Morgan's refinement calculus seamlessly.

Since Z cannot support implementation of the system directly, how to develop executable programs from Z specification has been a valuable research field.

The advantage of COOZ[14] is to specify a large scale software, but it does not support refinement by calculating and it needs proof in refinement. But the proof is very hard for OO specification, especially for the large and complex one. Thus its application is confined and it cannot be taken as a whole method for software development. Including refinement calculus into COOZ overcomes its disadvantage during design and implementation. The separation between design and implementation for and notation is removed as well. Then the software can be developed smoothly in the same frame. There is not correspondent object-oriented construct in the existing refinement calculus. The combination of COOZ and refinement calculus can build an object-oriented frame, in which the specification in COOZ is refined stepwise to code by calculus. In the paper, a development model is argued, which is based on COOZ and refinement calculus. The data refinement and operation refinement are analyzed by example; the two methods of operation refinement for OO formal specification is discussed briefly; the frame transition rule from COOZ to C++ is argued. On the implementation technology of refinement calculus, the data refinement calculator is constructed. Finally we argue an approach to data refinement based on data refinement calculus and program window inference.

2 Basic Concept

In this section, some basic concepts are introduced, which include refinement calculus, window inference, program window inference.

2.1 Refinement Calculus

The refinement calculus is based upon the weakest preconditions of Dijkstra, which views programs as predicate transformers, i.e., functions from postcondition to precondition. The refinement calculus extends the guard command of Dijkstra with specification statement. It is a wide spectrum language and a set of correctness preserving rules for deriving executable program from specification. The emphasis on refinement comes from the observation that it is more effective to develop a program and its correctness proof together, as opposed to attempting to verify a given program retrospectively. It is a calculus because the transformation rules calculate the refined program. It includes a specification statement in addition to the usual executable constructs. This integration of specification and execution in one language is the key to a smooth development process, since it allows a program to be de-

veloped by a series of transformations within a single language. The initial program is typically a specification statement, the final program contains only executable codes and the intermediate program is a mixture of the two.

One program statement that is particularly important in the refinement calculus is the *specification statement,* which provides a convenient way of embedding abstract specification into programs. Instead of writing specification indirectly as 'S where Pre $\Rightarrow$ $\square$ S $\square$ post', we write it directly as precondition-postcondition pair x : [pre, post]. The *frame* of this statement (x) is a variable list that can be updated. All other variables must remain unchanged. The specification statement is defined as:

$$[\![x : [\text{pre}, \text{post}]]\!]\text{P} \equiv \text{pre} \wedge (\forall x \cdot \text{post} \Rightarrow \text{P})$$

Following is an example of a specification statement, together with one possible refinement.

$$x : [\textbf{true}, x = \max(a, b)] \sqsubseteq \ \textbf{if}\ a \geq b x := a [\!] b \geq a x := b\ \textbf{fi}$$

2.2 Program Window Inference

The program window inference extends window inference to explicitly deal with program contexts. The goal is to provide better support for mechanizing refinement.

In program window inference the programs and predicates are separated and the explicit mechanisms for handling program context and program logic are argued. This allows one to reason more directly at the program level, without having to reduce everything to predicates.

Even simple programs can build up significant refinement context, hence, handling the context in effective way is an essential item for supporting program refinement. As most of the development of a program is done by refining its components, the program window inference theory provides good support for refinement of program component in context. Following is the way to handle preconditions.

Preconditions

For a (traditional) window:

$$\text{H} \models \{\text{P}\}\text{S} \sqsubseteq \{\text{P}\}\text{S}' \tag{1}$$

The equivalence program window is

$$\text{H};\ \textbf{pre}\ \text{P} \models \text{S} \sqsubseteq \text{S}' \tag{2}$$

where the notation **pre** is part of the syntax of program window, it represents the label of precondition.

Consider a selection command:

$$\{\text{P}\}\ \textbf{if}\ \text{g1} \rightarrow \text{S1}\ [\!]\ \text{g2} \rightarrow \text{S2}\ \textbf{fi} \tag{3}$$

In the refinement of S, it needs to make use of the precondition P and guard g1, thus these preconditions must become explicit, i.e., to replace (3) by

$$\{\text{P}\}\ \textbf{if}\ \text{g1} \rightarrow \{\text{P} \wedge \text{g1}\}\text{S1}\ [\!]\ \text{g2} \rightarrow \{\text{P} \wedge \text{g2}\}\text{S2}\ \textbf{fi} \tag{4}$$

This leads to the replication of information, which is partly undesirable if P is a large formula. So a new type window, the program window, is argued, which includes precondition along with ordinary hypotheses. In program window, the window opened for refinement of S1 in (3) is

$$\text{H};\ \textbf{pre}\ \text{P} \wedge \text{g1} \models \text{S1} \sqsubseteq \text{S1}' \tag{5}$$

Here the precondition context of the selection command, **pre** P, is automatically augmented by the guard g1 to form the precondition context for refinement of S1.

• **Window opening rules**

For a selection command (1), the window opening rule for refinement of S1 is:

$$\frac{\text{H; } \mathbf{pre}\ \text{P} \wedge \text{g1} \models \boxed{\text{S1}} \sqsubseteq \text{S1'}}{\text{H} \models \mathbf{if}\ \text{g1} \rightarrow \boxed{\text{S1}}\ \square\ \text{g2} \rightarrow \text{S2}\ \mathbf{fi}} \tag{6}$$

$$\mathbf{Pre}\ \text{P} \sqsubseteq\ \mathbf{if}\ \text{g1} \rightarrow \text{S1}' \square \text{g2} \rightarrow \text{S2}\ \mathbf{fi}$$

Following are other program window opening rules with precondition context.

• **Focus transformation rules**

Window opening rules express the transformation of a structure by transforming its components. It is also required to transform the focus directly, that is achieved by focus transformation rules. Usually, a focus transformation rule may have premises that establish the proof obligations to be discharged. The contexts accumulated by the window opening can be made use of discharging these obligations. For example, on introducing a selection command with guards g1, g2 for S, there is the rules:

$$\frac{\text{H; } \mathbf{pre}\ \text{P} \models \mathbf{pre}\ \text{g1} \vee \text{g2}}{\text{H; } \mathbf{pre}\ \text{P} \models \text{S} \sqsubseteq \mathbf{if}\ \text{g1} \rightarrow \text{S}\ \square\ \text{g2} \rightarrow \text{S}\ \mathbf{fi}} \tag{7}$$

where (H; **pre** P $\square$ **pre** g1 $\vee$ g2) is the rule premise that must be discharged.

As the premises and conclusion have the same form, (7) can be abbreviated by leaving the unchanged context of the program window implicit:

$$\frac{\mathbf{pre}\ \ \text{g1} \vee \text{g2}}{\text{S} \models \mathbf{if}\ \text{g1} \rightarrow \text{S}\ \square\ \text{g2} \rightarrow \text{S}\ \mathbf{fi}} \tag{8}$$

When the rule is used, the premises will be discharged with the same hypotheses as those for window being transformed.

3 Software Development Model

The development starts from the software specification in COOZ. By refinement calculus of state schema and operation schema in the class, the abstraction level is reduced stepwise until enough concrete specification is achieved.

3.1 Development Model

The model is based on COOZ frame (Fig.1). Firstly, to specify the system in COOZ (for example Fig.2), transform the operation schema into specification statement, then after the state schema is refined, the operation refined is calculated by data refinement calculus. According to the refinement law, the data refined specification statement is operation refined and the abstract program in COOZ frame is achieved. Finally, the abstract program is transformed into a programming language code, such as C++.

For example, a class schema *ClassStudent* is used for modeling the students in an exercise class. There are two kinds of students: one passed exercise, the other did not. For discussing simply, the following gives a simple form of the class schema *ClassStudent* (Fig.2).

Here just the registering method Enroll_OK is given, the others are omitted.

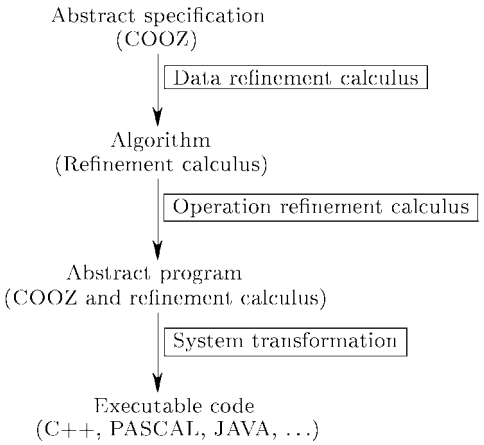

Fig.1. Development model 1.

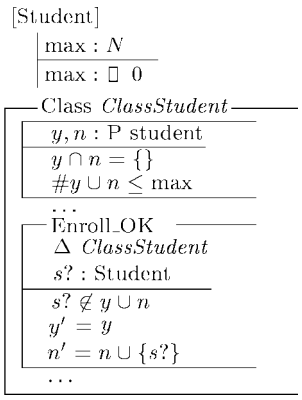

Fig.2. Class schema *ClassStudent*.

3.2 Data Refinement

By data refinement, abstract class A is refined by class C, noted as A □ C. But in Z and COOZ, the process needs to be proved[9].

As the refinement calculus is included, the operation schema in A can be noted as specification statement, after state schema is refined, the correct operation in C can be calculated and no more proof.

Morgan[13] extended the program definition by taking both specification and code as program. In the program, inexecutable code which will be refined is noted by specification statement. We extend that by allowing to include mathematical data type into the program, which is called abstract program.

There is a directed correspondent relation between specification statement and operation schema in COOZ (Fig.3).

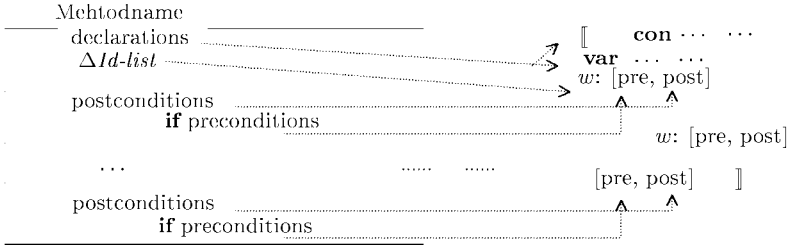

Fig.3. The correspondent relation of specification statement and operation schema.

In operation schema, the changeable variable is noted as Δ*Id-list*. For the predicates, the deference precondition is apart by a line and the precondition and postcondition are apart by a key word **if**. So the operation schema can be transformed into specification statement automatically. The specification statement is modified for different preconditions and the given postcondition respectively. Then in a specification statement, there are several pairs [pre, post].

For example, the operation schema in Fig.2, Enroll_OK, can be transformed into a specification statement:

$$n : [s? \notin y \cup n,\ n = n0 \cup \{s?\}] \tag{9}$$

where $n0$ denotes the reference to pre-state variable n. Next the data refinement of class schema in Fig.2 is analyzed.

The concrete representation for the sets given in the specification of Fig.2 will consist of two arrays, one for students, and the other for Boolean values, and a counter to say how many arrays is in use. It is intended that the values in the second array will be *true* for those who have done the exercises, and *false* for those who have not. The arrays are modeled by total functions whose domain is the index set (1..max). Then the class schema *ClassStudent* is refined as ClassStudent_1, in which the state schema is given as Fig.4:

cl: 1..max → Student ex: 1..max → Bool num: 0..max
((1..num) ◁ cl) ∈ (*N* ⤖ Student)

Fig.4. State schema of *ClassStudent*.

The concrete state invariant says that there will be no duplicates in the first num elements of the array of students.

The retrieve relation, relating the concrete and abstract states, is as follows:

$$R \triangleq (y = \{i : 1..\text{num} | (\text{ex}\ i) = \text{true} \cdot (\text{cl}\ i)\}) \wedge \{i : 1..\text{num} | (\text{ex}\ i) = \text{false} \cdot (\text{cl}\ i)\})$$

Following the data refinement a: [pre, post] $\prec$ c: [sim pre, sim post], and with R and (9), the operation on state of Fig.4 can be calculated. Here the simulation sim is defined as: sim $Q \triangleq \exists a \bullet R \wedge Q$.

$$\begin{aligned} & n : [s? \notin y \cup n,\ n = n0 \cup \{s?\}] \prec \\ & \text{cl}, \text{ex}, \text{num} : [\exists n \bullet R \wedge s? \notin y \cup n, \exists n \bullet R \wedge n = n0 \cup \{s?\}] \\ & = \text{cl}, \text{ex}, \text{num} : [(\text{num} < \text{max}) \wedge (s? \notin \{1..\text{num} \bullet \text{cl}\ i\}), (\text{num} = \text{num0} + 1) \wedge \\ & (\text{cl} = \text{cl0} \oplus (\text{num} \mapsto s?)) \wedge (\text{ex} = \text{ex0} \mapsto (\text{num} \mapsto \text{false}))] \end{aligned} \tag{10}$$

From (10), we can get the refined operation schema in ClassStudent_1 easily. In Section 4, we will debate data refinement in detail.

3.3 Operation Refinement

For example, the data refined operation schema, i.e., the correspondent specification statement (10) is refined into executable code easily by refinement law of Morgan[13]:

⊑num,cl[num+1], ex[num+1] = num+1, s, false
⊑num = num+1;
  cl[num+1] = s
  ex[num+1] = false

4 Data Refinement Calculus

4.1 Data Refinement

Data refinement can be viewed as a special case of program refinement, in which the abstract local variables of a program are replaced by the concrete set of variables, while the structure of the program remains largely unchanged. It is also useful to have a direct data refinement relation between components of the abstract program and concrete program.

Early definition of data refinement assumed a functional relationship from the concrete variables c to the abstract variables a, so that each state of the concrete program could be mapped into some state of the original abstract program. This allowed several distinct concrete states to correspond to the same abstract state, which is necessary, since data refinement often introduces redundancy to improve the efficiency of some operations. However, some useful data refinements could not be described by a functional relationship, so data refinement was later generalized to use a relation between the abstract and concrete variables.

More recently still, the relational approach has been generalized to allow the abstract and concrete state spaces to be related to an arbitrary program[15]. This program is called *simulation* if it converts from the abstract state space to the concrete, or *co-simulation* if it converts from the concrete state space to the abstract.

In the following discussion we assume that the state space of program S is variable to be refined (a) with some global variables (g), and the state space of program S$'$ is the concrete variables (c) with g, where a, c and g are all disjoint.

We choose any predicate transformer sim that takes predicate on the variables a, g to predicates on the variables c, g. For programs P and P$'$, P is data refined by P$'$, written as P $\prec$ P$'$.

$$\text{Data Refinement Definition 1}: \ \mathrm{P} \prec \mathrm{P}' \text{ iff } \mathrm{sim};\ \mathrm{P} \sqsubseteq \mathrm{P}';\ \mathrm{sim} \tag{11}$$

where the operator ';' is functional compositional (of predicate transformers). In fact the above sim is a co-simulation.

If the sim is a simulation, noted as sim*, then the data refinement is defined as:

$$\text{Data Refinement Definition 2}: \ \mathrm{P} \prec \mathrm{P}' \text{ iff } \mathrm{P};\ \mathrm{sim}^* \sqsubseteq \mathrm{sim}^*;\ \mathrm{P}' \tag{12}$$

Given the abstraction relation AI between abstract and concrete data structures, for any predicate φ over abstract variables, the semantics of sim and sim* can be defined in the weakest precondition as follows:

$$[\![\mathrm{sim}]\!]\varphi \triangleq \exists a \bullet \mathrm{AI} \wedge \varphi \tag{13}$$

$$[\![\mathrm{sim}^*]\!]\varphi \triangleq \forall a \bullet \mathrm{AI} \Rightarrow \varphi \tag{14}$$

In practice, it is useful to place some restrictions on sim to ensure that data refinement is distributed through various program constructors and thus leaves the structure of the abstract program unchanged. For example, if the co-simulation sim is disjunctive and strict, then the data refinement relation $\prec$ satisfies the following rule:

$$(\mathrm{S1} \prec \mathrm{S1}') \wedge (\mathrm{S2} \prec \mathrm{S2}') \Rightarrow \mathrm{S1}; \mathrm{S2} \prec \mathrm{S1}'; \mathrm{S2}' \tag{15}$$

4.2 Calculation of Data Refinement

4.2.1 Program Block

Since the scope of the local variable is a block statement, data refinement is modeled as a transformation of the whole block. In this way, data refinement becomes a special case of algorithm refinement: the refinement is just on the whole block.

The block is noted as:

$$|[\mathbf{var}\ a \mid \mathrm{Init} \bullet \mathrm{P}]| \tag{16}$$

which is the target of data refinement. **var** a is a local variable added into the program and is initialized according to the initialization predicate Init.

The semantics of the block is given in weakest precondition: for any predicate φ not containing free a,

$$[\![|[\mathbf{var}\ a \mid \mathrm{Init} \bullet \mathrm{P}]|]\!]\varphi \triangleq \forall a \bullet \mathrm{Init} \Rightarrow [\![\mathrm{P}]\!]\varphi \tag{17}$$

4.2.2 Data Refinement Calculator

For data refinement of the block, we assume that the abstract and the concrete variables are related by abstract relation AI. The data refinement calculator $\mathrm{D_{AI}}$ for predicates and D_{AI} for program statements are introduced, such that

$$|[\mathbf{var}\ a \mid \mathrm{Init} \bullet \mathrm{P}]| \sqsubseteq |[\mathbf{var}\ \mathrm{c}\ |\mathrm{D_{AI}}(\mathrm{Init}) \bullet D_{\mathrm{AI}}(\mathrm{P})]| \tag{18}$$

For a predicate φ, $\mathrm{D_{AI}}$ and its dual calculator $\mathrm{D^*_{AI}}$ are defined as:

$$\mathrm{D_{AI}}(\varphi) \triangleq \exists\, a \bullet \mathrm{AI} \wedge \varphi \tag{19}$$

$$\mathrm{D^*_{AI}}(\varphi) \triangleq \forall\, a \bullet \mathrm{AI} \Rightarrow \varphi \tag{20}$$

From Subsection 3.1 and following the concept of dual and adjoint of the predicate transformer in data refinement[16], we have:

$$\mathrm{P} \prec \mathrm{P}' \text{ iff } \mathrm{sim}; \mathrm{P}; \mathrm{sim}^* \sqsubseteq \mathrm{P}' \tag{21}$$

Thus, (sim; P; sim*) is defined as the least (most abstract) data refinement for statement P such that:

$$\mathrm{sim}; \mathrm{P}; \mathrm{sim}^* \sqsubseteq D_{\mathrm{AI}}(\mathrm{P}) \tag{22}$$

4.3 Application of Data Refinement Calculator

For different program structures, the calculator for statements can be defined recursively over the structure of program notation and reflects data refinement theorems for the corresponding program statement. According to Ruksenas *et al.*[17], following gives some feature of data refinement calculator.

• Specification statement

For specification statement $\mathrm{P} = a, g : [\mathrm{pre}, \mathrm{post}]$, we can get the following formula from (22):

$$D_{\mathrm{AI}}(a, g : [\mathrm{pre}, \mathrm{post}]) \sqsubseteq c, g : [\mathrm{sim\ pre}, \mathrm{sim\ post}] \tag{23}$$

Thus we get the data refinement for P:

$$a, g : [\mathrm{pre}, \mathrm{post}] \prec c, g : [\mathrm{sim\ pre}, \mathrm{sim\ post}] \tag{24}$$

For predicate φ, sim φ can be taken as the same as $D_{\mathrm{AI}}(\varphi)$, so are sim$^*\varphi$ and $\mathrm{D^*_{AI}}(\varphi)$. Thus, (24) can be written as

$$a, g : [\mathrm{pre}, \mathrm{post}] \prec c, g : [\mathrm{DAI(pre)}, \mathrm{DAI(post)}] \tag{25}$$

This rule is particularly useful, since it allows the correct specification statement to be calculated from the abstract specification statement. In fact, (24) is the same as the result of Morgan *et al.*[15].

• Assignment and context statement

For a non-deterministic assignment $\mathrm{post}[a, g \backslash a', g']$, its data refinement is calculated as:

$$D_{\mathrm{AI}}(\mathrm{post}[a, g \backslash a', g']) = \forall a \bullet \mathrm{AI} \Rightarrow (\exists a' \bullet \mathrm{AI}[a, g \backslash a', g'] \wedge \mathrm{post}[a, g \backslash a', g'])$$

Th context includes assertion $\{\varphi\}$ and assumption $[\varphi]$. An assertion $\{\varphi\}$ in a program asserts that condition φ holds at that point in the program. An assumption $[\varphi]$ represents our anticipation that φ holds at that point. (However, the validity of the anticipation needs to be demonstrated at some point). For above context, D_{AI} is defined as:

$$D_{\mathrm{AI}}(\{\varphi\}) \triangleq \{\mathrm{D_{AI}}(\varphi)\}$$

$$D_{\mathrm{AI}}([\varphi]) \triangleq [\mathrm{D_{AI}}(\varphi)]$$

• Sequential composition

The data refinement calculator distributes on sequential composition:

$$D_{\mathrm{AI}}(S1; S2) = D_{\mathrm{AI}}(S1); D_{\mathrm{AI}}(S2)$$

It is the same as (15).

• **Condition, alternative, and loop statements**

The data refinement calculator can distribute into a condition statement, but in this case an assumption statement must be added to one branch of concrete condition. A guard command g → S is data refined as:

$$D_{\mathrm{AI}}(\mathrm{g} \to \mathrm{S}) = \mathrm{D}_{\mathrm{AI}}(\mathrm{g}) \to [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(S)$$

The assumption $[\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]$ is also needed for distribution of data refinement over alternative and loop statements.

Thus for condition statement:

$$D_{\mathrm{AI}}(\textbf{if g then S1 else S2 fi}) = \textbf{if } \mathrm{D}_{\mathrm{AI}}\ (\mathrm{g}) \textbf{ then } [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(\mathrm{S1}) \textbf{ else } D_{\mathrm{AI}}(\mathrm{S2}) \textbf{ fi}$$

For alternative statement:

$$\begin{aligned} &D_{\mathrm{AI}}(\textbf{if } (\mathrm{g1} \to \mathrm{S1}) \,[]\, \cdots \,[]\, (\mathrm{g}n \to \mathrm{S}n)\textbf{fi}) \\ &\quad = \textbf{if } \mathrm{D}_{\mathrm{AI}}(\mathrm{g}) \to [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(\mathrm{S}) \,[]\, \cdots \,[]\, \mathrm{D}_{\mathrm{AI}}(\mathrm{g}) \to [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(\mathrm{S}) \textbf{ fi}. \end{aligned}$$

For loop statement:

$$\begin{aligned} &D_{\mathrm{AI}}(\textbf{do}(\mathrm{g1} \to \mathrm{S1}) \,[]\, \cdots \,[]\, (\mathrm{g}n \to \mathrm{S}n)\textbf{od}) \\ &\quad = \textbf{do}\, \mathrm{D}_{\mathrm{AI}}(\mathrm{g}) \to [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(\mathrm{S}) \,[]\, \cdots \,[]\, \mathrm{D}_{\mathrm{AI}}(\mathrm{g}) \to [\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})]; D_{\mathrm{AI}}(\mathrm{S}) \textbf{ od} \end{aligned}$$

• **Block**

For a block the data refinement calculator distributes over inner block as follows:

$$D_{\mathrm{AI}}(|[\textbf{var b} \mid \mathrm{Init} \bullet \mathrm{Q}]|) = |[\textbf{var b} \mid \mathrm{D}^*_{\mathrm{AI}}(\mathrm{Init}) \bullet D_{\mathrm{AI}}(\mathrm{Q})]|$$

5 Program Window Inference with Data Refinement

In performing refinement of a component of a program, the context of the component is important. Window inference and program window inference introduced above provide an excellent approach to handling such contextual information. But the data refinement is not included in the approach. Data refinement is harder to deal with in a refinement tool than ordinary algorithmic refinement, since data refinement usually has to be done on a large program component at once.

Since the scope of the local variable is a block statement, data refinement is modeled as a transformation of the whole block. In this way, data refinement becomes a special case of algorithm refinement: the refinement is just on the whole block.

In this section, we present an approach to handling data refinement with program window inference, which is based on data refinement calculator introduced in Subsection 3.2.

5.1 Window Opening Rule

The data refinement is just on the block, thus only one window opening rule is required for the block. As the program cannot operate on both the abstract and the concrete states at the same time, hence a data refinement transformation must replace all occurrences of the abstract variable in one step.

For data refinement on the block (18), we have the rule:

$$\frac{\mathrm{H};\ \textbf{pre}\ \mathrm{P}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{D}_{\mathrm{AI}}(\mathrm{Init});\ \textbf{lval}\ \mathrm{L}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \textbf{Var};\ \textbf{inv}\ \mathrm{Inv}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \mathrm{T}' \wedge \mathrm{AI} \vdash \boxed{\mathrm{P}} \prec D_{\mathrm{AI}}\ (\mathrm{P})}{\mathrm{H};\ \textbf{pre}\ \mathrm{P};\ \textbf{lval}\ \mathrm{L};\ \textbf{inv}\ \mathrm{Inv} \vdash |[\textbf{var}\ \mathrm{a} : \mathrm{T} \mid \mathrm{Init} \bullet \boxed{\mathrm{P}}\ ]| \sqsubseteq |[\textbf{var}\ \mathrm{c} : \mathrm{T}' \mid \mathrm{D}_{\mathrm{AI}}\ (\mathrm{Init}) \bullet D_{\mathrm{AI}}\ (\mathrm{P})]|} \quad (26)$$

Here AI is the abstract invariant representing the relation between the abstract and concrete variables. With this rule, the block transformation focuses on calculation of the concrete commands, while the other complex predicates are collected into the **pre** and **inv** contexts, such as initialization for concrete variable $\mathrm{D_{AI}}$(Init) and AI.

The symbol $\prec$ represents data refinement relation between abstract and concrete commands (see Subsection 3.1), and it is reflective and transitive. For getting the same relation in subwindow, ($\boxed{\mathrm{P}} \prec D_{\mathrm{AI}}(\mathrm{P})$) can be written as $D_{\mathrm{AI}}(\boxed{\mathrm{P}}) \sqsubseteq \mathrm{P}'$, and (26) is written as:

$$\frac{\mathrm{H};\ \mathbf{pre}\ \mathrm{P}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{D_{AI}}(\mathrm{Init});\ \mathbf{lval}\ \mathrm{L}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \mathrm{Var};\ \mathbf{inv}\ \mathrm{Inv}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \mathrm{T}' \wedge \mathrm{AI} \vdash D_{\mathrm{AI}}(\boxed{\mathrm{P}}) \sqsubseteq \mathrm{P}'}{\mathrm{H};\ \mathbf{pre}\ \mathrm{P};\ \mathbf{lval}\ \mathrm{L};\ \mathbf{inv}\ \mathrm{Inv} \vdash |[\mathbf{var}\ \mathrm{a}:\ \mathrm{T} \mid \mathrm{nit} \bullet \boxed{\mathrm{P}}]| \sqsubseteq |[\mathbf{var}\ \mathrm{c}:\ \mathrm{T}' \mid \mathrm{D_{AI}}(\mathrm{Init}) \bullet \mathrm{P}']|} \quad (27)$$

This rule's form is the same as the algorithm refinement rule.

5.2 Focus Transformation Rules

For different program structures, the calculator for statements can be defined and reflects data refinement theorems for the corresponding program statement. The performance of data refinement calculator can be modeled with focus transformation rule by using the inv context information as the obligation to be discharged.

For example, on sequence composition structure S1; S2, the rule is written as:

$$\frac{\mathrm{H};\ \mathbf{pre}\ \mathrm{P}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{D_{AI}}(\mathrm{Init});\ \mathbf{lval}\ \mathrm{L}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \mathbf{Var} \vdash \mathbf{inv}\ \mathrm{Inv}[\mathrm{c}\backslash\mathrm{c}'] \wedge \mathrm{c} \in \mathrm{T}' \wedge \mathrm{AI}}{\mathrm{S1};\ \mathrm{S2} \prec D_{\mathrm{AI}}(\mathrm{S1});\ D_{\mathrm{AI}}(\mathrm{S2})} \quad (28)$$

Using program window inference, some calculation can be simplified. For example, for guard command

$$D_{\mathrm{AI}}(\{\mathrm{P}\};\ \mathrm{g} \to \mathrm{S}) = \{\mathrm{D_{AI}}(\mathrm{P})\};\ \mathrm{D_{AI}}(\mathrm{g}) \to ([\mathrm{D}^*_{\mathrm{AI}}(\mathrm{g})];\ D_{\mathrm{AI}}(\mathrm{S}))$$

in program window inference, it can be written as:

$$D_{\mathrm{AI}}(\mathrm{pre\ P} \wedge \mathrm{g} \vdash \mathrm{S}) = \mathrm{D_{AI}}(\mathrm{pre\ P} \wedge \mathrm{g}) \vdash D_{\mathrm{AI}}(\mathrm{S})$$

5.3 Comparison with Algorithm Refinement Rule

In this paper, data refinement is handled as a special algorithm. But it is different from window opening and focus transformation rules.

• **Window opening rule**

For data refinement there is just one window opening rule that is on the variable declaration block. The focus is on all commands in the block, not their component, since the program cannot operate on both the abstract and the concrete states at the same time, hence a data refinement transformation must replace all occurrences of the abstract variable in one step.

When focussing inside the block, the abstract invariant AI (noting the relation of the abstract and concrete variables) and the predicates transformed such as $\mathrm{D_{AI}}$(Init) are augmented into the context of the new window.

• **Focus transformation rule**

For algorithm refinement, the focus transformation follows the refinement calculus rule of Morgan[13], which refines the statement into different constructs (such as iteration, alteration, loop).

For data refinement, the focus transformation does not change the structure of the command. It just calculates the concrete command from the abstract command by the data refinement calculator.

6 Conclusion and Future Work

In the formal specification of each class, the data are described with abstract data structure of COOZ, and the operations are defined with pre and post conditions. During refinement, we first replace the abstract data structure with concrete data structure in programming language. This is called data refinement. Then, we select the appropriate algorithm satisfying corresponding pre and post conditions to implement each operation. This is called operation refinement. Complex operations may need stepwise refinements. Sometimes, new operations must be added to decrease the complexity. The new operations are the internal operations of class. The refinement process is stepwise, and every refinement generates a new, more detailed specification.

In fact, here just the frame and the theory basis of the development models are researched. Many concrete technologies need to be studied. We have argued an approach for data refinement, which is based on data refinement calculus and program window inference. The approach supports the calculation style of data refinement: a concrete program can be automatically constructed from the abstract one. The program window inference is an effective way to manage complexity during refinement. Managing and providing access to program context can lead to simpler refinement with less replication.

One of the advantages of formal method is that it promotes the degree of software automation. The refinement of data can be automatically done. In the refinement of operations, the algorithm must be designed or selected. So, the operation refinement cannot be done automatically. However, it can be done with semiautomatic strategy. The research and application of automation technique are worthy of further enhancing.

The model presented here requires the support of tools, which include prototyping tools, management tools for class library, and refinement tools for formal specification. In fact, it is difficult for any software methods to be of practical use without the support of integrated CASE. The further work is to provide necessary tools and appropriate environment. The work is being done now.

References

[1] de Bakker J W *et al.* (eds.) In *Proc. REX Workshop on Stepwise Refinement on Distributed Systems,* Lecture Notes in Computer Science 430, Springer-Verlag, 1989.

[2] Hoare C A R, He J. The Weakest Prespecification. *Fund. Inform. IX* 1986, pp.51–84.

[3] Jones C B, Shaw R C, Denvir T (eds.). In *5th Refinement Workshop in Computing,* Springer-Verlag, 1992.

[4] Back R J R. On the correctness of refinement in program development [dissertation]. Report A-1978-4. Department of Computer Science, University of Helsinki, 1978.

[5] Morris J M. A theoretical basis of stepwise refinement and programming calculus. *Science of Computer Programming,* 1987, 9(2): 287–306.

[6] Morgan C C. The specification statement. *ACM Transaction on Programming Language and Systems,* July 1988, 10(3): 403–419.

[7] Back R J R, von Wright J. Refinement calculus, Part I: Sequential programs. In *REX Workshop for Refinement of Distributed Systems,* Lecture Notes in Computer Science 430, Nijmegen, The Netherlands, Springer-Verlag, 1989.

[8] Gardiner P H B, Morgan C. A single complete rule for data refinement. *Formal Aspects of Computing,* 1993, 5(4): 367–382.

[9] Spivey J M. The Z Notation: A Reference Manual. Prentice-Hall, International Series in Computer Science, 2nd Edition, 1992.

[10] Stepeney S *et al.* More powerful Z data refinement. In *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users,* Bowen J P, Hinchey M G (eds.), September 1998, Proceedings, Lecture Notes in Computer Science 1493, Springer-Verlag, 1998.

[11] Nellson D S. From Z to C: Illustration of a rigorous development method [dissertation]. PRG-79, Oxford University, Computing Laboratory, February, 1990.

[12] King S. Z and the refinement calculus. In VDM and Z – Formal Methods in Software Development, Bjϕrner D *et al.* (eds.), Lecture Notes in Computer Science 428, VDM-Europe, Springer-Verlag, 1990,

pp.164–188. Also published as a Technical Monograph PRG-79, Oxford University, Computing Laboratory, February, 1990.
[13] Morgan C C. Programming from Specifications. Prentice-Hall, International Series in Computer Science, 2nd Edition, 1994.
[14] Yuan Xiaodong, Hu Deqiang, Xu Hao *et al.* COOZ: A complete object-oriented extension to Z. *ACM Software Engineering Notes,* 1998, 23(4): 78–81.
[15] Gardiner P H B, Morgan C. Data refinement of predicate transformers. *Theoretical Computer Science*, 1991, 87: 143–162.
[16] J. von Wright. A lattice-theoretical base for program refinement [dissertation]. Abo Akademi University, SF-20500 Turku, Finland, Sept., 1990.
[17] Rimvydas Ruksenas. A tool for data refinement. Technical Report, TUCS — Turku Centre for Computer Science, Number TUCS-TR-119, August, 1997.

WANG Yunfeng received his Ph.D. degree from Department of Computer Science and Technology, Nanjing UIniversity in 2000. He is now working in Meteorology College, PLA University of Science and Technology. His research interests include formal methods and object-oriented technology.

PANG Jun received his M.S. degree from Department of Computer Science and Technology, Nanjing University in 2000. He is now a Ph.D. candidate in Department of Software Engineering, National Center for Mathematics and Computer Science (CWI), Netherlands. His interests include process algebra, protocol verification, Z & refinement calculus and object-oriented technology.

YANG Zhaohui received his B.S. degree from Department of Computer Science and Technology at Nanjing University in 2000. He is now an M.S. candidate of Department of Computer Science and Technology, Nanjing University.

ZHENG Guoliang is a professor in Department of Computer Science and Technology, Nanjing University. He received his B.S. degree in computer science from Nanjing University in 1961. His main research area is software engineering.