An Integrated Refinement Calculus For Z

Bing Wu[1], D.R.W. Holton[2], and Luming Lai[3]

[1] B.Wu@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/∼bwu/
[2] D.R.W.Holton@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/∼drwholton
[3] L.M.Lai@scm.brad.ac.uk,
Home page: http://www.personal.comp.brad.ac.uk/∼lmlai
University of Bradford, Bradford, BD7 1DP, West Yorkshire, UK

Abstract. After more than twenty years of research, formal methods are beginning to be used in industry. Despite theoretical progress and numerous case studies and pilot projects, adoption of formal methods by organizations has been slow, partly due to concerns about their cost, lack of tool support, and the skills required for their application.
The paper proposes a more practical formal software development method by integrating the Refinement Calculus with Z, and presents a software development environment in which software can be formally specified and refined to program code. We present the concept and methodology for refining a Z specification to C code. Finally, potential theoretical and practical problems of implementing such a mechanised software development method are discussed.

Keywords: Formal methods, Z, Refinement Calculus, C, Mechanised software development

1 Introduction

Many of the most serious problems in software design and implementation result from imprecise, ambiguous, incomplete, misunderstood, or just plain incorrect statements in requirements specifications. Describing large, complex systems with thousands or even millions of properties is extremely difficult using natural language.

Formal methods (FM) consist of a set of techniques and tools that are based on mathematical modeling and formal logic and which are employed to specify and verify requirements and designs for computer systems - both hardware and software. Early successes in the use of FM were more frequently obtained in the design of computer hardware, and this domain remains an important application area for FM [10, 15]. Moreover, the growing criticality and complexity of software has led to increased interest in applying FM to software specification and design as well [12, 27].

Z [11, 23] is a well established specification language that has a distinguishing mechanism of modularization: the schema calculus. It allows us to formalise individual requirements separately and join them together by schema operators. In the following example, we describe this by specifying a sales information system. The same example can be found in [9].

Example 1. *The sales database is a sample system which records a company's sales business information system. The Z specification of a sales database includes a global constant: TeamSize, which gives the number of people in the company sales team. This is introduced below by meaning of an axiomatic description*

$$| \; TeamSize : \mathbb{N}$$

A number of messages are needed to show the system information. These are all members of the set Report:

$$\begin{array}{rcl} Report & ::= & `Okay' \\ & | & `Unknown\ salesperson' \\ & | & `Input\ not\ confirmed'. \end{array}$$

We describe the state space of the system as a schema named SalesDB. The sales database is represented by three arrays which are modeled by functions from the set $\{1 \,..\, TeamSize\}$ to the set $\mathbb{N}_1$ of integers and three state variables.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SalesDB}} \\
\hline
sales, comm, bonus : 1 \mathinner{..} TeamSize \rightarrow \mathbb{N} \\
tot, ave, thresh : \mathbb{N} \\
\hline
tot = sumarray(sales) \\
ave = tot \ \mathbf{div} \ TeamSize \\
thresh = (2 * ave) \ \mathbf{div} \ 3 \\
\hline
\end{array}$$

In this state schema, the array sales keep track of the running total of the sales made by each team member and the array comm and bonus keep, respectively, the amount of commission and the bonus that each team member has earned. The variable tot keeps track of the combined value of all the sales made by the whole team and ave is the result of dividing tot by the size of the team. The variable thresh represents the value of the sales made by a sales person on which he or she does not receive commission. The function sumarray returns the sum of an array of numbers.

The operation AddSales updates the sales database when a sales person makes a sale. The variable ref? is the reference number of the team member whose records are to be updated. The variables amnt? and conf? represent the value of the sales that sales person has made. To avoid typing errors being made by the user, this number has to be typed twice and the two values are compared. The sales array and the total of the sales are updated.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddSales}} \\
\hline
\Delta SalesDB \\
ref?, amnt?, conf? : \mathbb{N} \\
\hline
ref? \in 1 \mathinner{..} TeamSize \\
amnt? = conf? \\
sales' = sales \oplus \{ref? \mapsto sales[ref?] + amnt?\} \\
tot' = tot + amnt? \\
comm' = comm \\
bonus' = bonus \\
\hline
\end{array}$$

We define the schema Success to indicate the successful completion of a schema operation in this way:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Success}} \\
\hline
rep! : Report \\
\hline
rep! = \text{`}Okay' \\
\hline
\end{array}$$

The schema AddSalesOkay is the same as AddSales except a suitable message is printed out to indicate the successful completion of the AddSales operation.

$$AddSalesOkay \mathrel{\widehat{=}} AddSales \land Success.$$

The Z schema calculus makes the specification of industrial-scale systems practical. The semi-graphical schema notation also makes Z simpler and more understandable than some other formal methods, such as VDM [13] and the B-Method [28], which are keyword-based.

The success of Z is evident: many case studies [11] have already been developed, some of which involve industrial applications [7].

2 The Problem

However, Z is different from VDM and B which support development methods based on the notion of refinement. If we compare Z with VDM and B, we can see one important difference between them: VDM and B are refinement development methods, whereas Z is primarily a specification language. The result of this is that specification formulae in VDM and B use keywords like a programming language and are syntactically more complicated than Z schemas. In addition, with refinement based development in mind, VDM and B also separate the precondition from the postcondition.

B uses the Abstract Machine Notation (AMN) to model systems. AMN is a state-based formal specification language in the same school as VDM and Z. An Abstract Machine comprises a state together with operations on that state. In a specification and a design of an Abstract Machine the state is modeled using notions like sets, relations, functions, sequences etc.. In an implementation of an abstract machine the state is again modeled using a set-theoretical model. The operations are described using a pseudo-programming notation that is a subset of AMN. In comparison with B, the refinement of VDM specifications ends with more concrete VDM specifications.

Unlike VDM and B, Z lacks a well-defined and provably correct method of moving from the specification phase to the later stages of program development. This problem is now widely recognized [14, 27]. But the proposed development methods have yet to be formalized.

To make the refinement of industrial-scale system practical, there are basically two aspects of the problem to be tackled in Z. First, although Z is a well established specification language and the Refinement Calculus was developed for Z by Morgan [17] and others [1, 18] some years ago, the integration of both is not as simple as originally thought. There are some fundamental problems to be solved for a smooth integration. For example, the Refinement Calculus is based on the weakest precondition semantics but Z on set theory; the specification statements in the Refinement Calculus use pre- and postconditions but Z uses the schema calculus which puts abstraction above everything else. A more serious problem is that the conjunction and disjunction operators in the schema calculus are not monotonic with respect to the refinement relation, as proved in [25], and it is not clear how the other schema operators can be introduced into the Refinement Calculus. Therefore, further research needs to be done to make the refinement laws applicable to schemas with inherited properties and operations.

Secondly, in the Refinement Calculus, although the refinement laws are simple and elegant, the proofs of refinement steps are tedious and very time consuming. Tools are urgently needed if such formal methods are to be applied in practice. However, none of the existing tools support refinement using Z.

3 Methodology

The main objective of our research is to develop a mechanised software development method which supports system development from Z specification to final C code.

The steps of this research project are as follows. Firstly, in Section 3.1, we investigated how to adapt Z schemas to suit the refinement calculus which was developed for the *pre*− and *postcondition* specification statements. The introduction of *pre*− and *postconditions* into Z schemas is investigated, resulting in slightly modified Z schemas. We call this modified Z. Schema operations are defined for modified Z which are monotonic with respect to the refinement relation except the disjunction operator.

Secondly, in Section 3.2, we investigated the theoretical foundations for the integration of Z and the refinement calculus which include the semantic properties of Z schemas and their relationships with the refinement laws; A weakest precondition semantics for Z has already been developed by Cavalcanti and Woodcock [5], which provides us a good starting point because the Refinement Calculus is also based on the weakest precondition semantics. A weakest precondition semantics is given for modified Z and C, that allows the justification of the refinement rules introduced in the refinement calculus and also provides a basis for other laws to be derived.

3.1 Conversion Calculus: From Z to Modified Z

Z, as a "pure" specification language, tries to distance itself from implementation consideration. It uses abstract data types, allows the introduction of new signatures anywhere, and does not care about the pre- and postconditions of operations. If we want to use Z through the development of software, Z needs to be modified.

The signature of a Z schema explicitly lists all the variable in its state space and the schema places no restriction on variables outside its signature. This means if a schema S_1 does not include a variable x but a schema S_2 changes x, then their conjunction $S_1 \wedge S_2$ can still change x. We call this an *open-world view* [24].

However, unlike a schema, variables in a specification statement $w : [pre, post]$ which do not appear in its frame w cannot be changed by the statement. We call this a *closed-world view* [24].

An open-world view is appropriate while specifying components that are to be composed together to define new operations, while a closed-world view is appropriate once an operation has been completely specified and is being refined, since during the refinement progresses it is more convenient to describe what does change rather than what does not in a state space with a large number of variables.

Since, in this project, we are interested in incremental composition of programs, it is useful to define a specification statement with an open-world view. To solve this problem, in [33] we propose an open-world view specification, modified Z, which splits the predicates of Z into *pre−* and *postcondition* as well as keeping the Z incremental style of developing specifications.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Schema}} \\
\hline
\textit{Declaration Part} \\
\hline
\textit{Precondition Part} \\
\hline
\textit{Postcondition Part} \\
\hline
\end{array} \qquad [d \mid pre \mid post]$$

So the first step of system development is converting the Z specification to the modified Z specification by applying conversion laws at the very beginning of the refinement development. We present some conversion laws below (a more detailed account can be found in [31]). Most of the conversion laws are based on those of [14, 4].

The basic conversion law converts any Z schema to the corresponding modified Z schema. It is the most useful law in the conversion calculus. If S is a Z schema, ds declares the state variables of S; $di?$, the declaration of the input variables of S; $do!$, the output variables; p, the predicates of S, then we have the following basic conversion law.

Law 1 (Basic Conversion).

$$\begin{aligned}
&[\Delta S;\ di?;\ do! \mid p] \\
= &[d \mid inv \wedge \exists\, ds';\ do! \bullet inv' \wedge p \mid inv' \wedge p]
\end{aligned}$$

where $S \mathrel{\widehat=} [ds \mid inv]$ *and* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$.

The Z schema *AddSales* in Example 1 can be converted into modified Z schema by applying the above.

$$AddSales = \{Basic\ Conversion\ Law\ 1\}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddSales}} \\
\hline
sales, sales', comm, comm', bonus, bonus' : 1 \mathrel{..} TeamSize \rightarrow \mathbb{N} \\
tot, tot', ave, ave', thresh, thresh' : \mathbb{N} \\
ref?, amnt?, conf? : \mathbb{N} \\
\hline
tot = sumarray(sales) \\
ave = tot\ \mathbf{div}\ TeamSize \\
thresh = (2 * ave)\ \mathbf{div}\ 3 \\
ref? \in 1 \mathrel{..} TeamSize \\
amnt? = conf? \\
\hline
tot' = sumarray(sales') \\
ave' = tot'\ \mathbf{div}\ TeamSize \\
thresh' = (2 * ave')\ \mathbf{div}\ 3 \\
sales' = sales \oplus \{ref? \mapsto sales[ref?] + amnt?\} \\
tot' = tot + amnt? \\
comm' = comm \\
bonus' = bonus \\
\hline
\end{array}$$

Z Schemas which specify operations that do not change the state, have the form

$$[\Xi S;\ di?;\ do! \mid p].$$

They can be written as:

$$[\Delta S;\ di?;\ do! \mid p \wedge s1' = s1 \wedge \cdots \wedge sn' = sn]$$

where $\alpha ds = s1 \cup \cdots \cup sn$ and $s1, \cdots, sn$ are state components of schema S. Since schemas of this form occur very frequently in Z specifications, we present an additional conversion law as below.

Law 2 (Basic Conversion).

$$
\begin{array}{l}
\quad [\Xi S;\ di?;\ do! \mid p] \\
= [d \mid inv \land \exists\, do! \bullet p[\alpha ds/\alpha ds'] \mid p]
\end{array}
$$

where $S \mathrel{\widehat=} [ds \mid inv]$ *and* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$.

To demonstrate the above law, we need to introduce more operations of the sales database in Example 2.

Example 2. *The operation AddSales can go wrong in two different ways. The first error occurs when* $ref?$ *is not in the set* $1 \mathrel{..} TeamSize$. *The schema UnknownSalesperson specifies the circumstance.*

$$
\begin{array}{|l}
\hline
\textit{UnknownSalesperson} \\
\Xi SalesDb \\
ref? : \mathbb{N} \\
rep! : Report \\
\hline
ref? \notin 1 \mathrel{..} TeamSize \\
rep! = \text{`}\textit{Unknown salesperson}\text{'} \\
\hline
\end{array}
$$

The second error comes with the schema AddSales when $amnt?$ *differs from* $conf?$. *This circumstance is specified by the schema NotConfirmed.*

$$
\begin{array}{|l}
\hline
\textit{NotConfirmed} \\
\Xi SalesDB \\
amnt?, conf? : \mathbb{N} \\
rep! : Report \\
\hline
amnt? \neq conf? \\
rep! = \text{`}\textit{Input not confirmed}\text{'} \\
\hline
\end{array}
$$

The specification of the total operation is given by the schema DoAddSales:

$$DoAddSales \mathrel{\widehat=} AddSalesOkay \lor UnknownSalesperson \lor NotConfirmed.$$

The schemas *UnknownSalesperson* and *NotConfirmed* can be converted into the modified Z schemas by the above law.

$$UnknownSalesperson = \{Basic\ Conversion\ Law\ 2\}$$

$$
\begin{array}{|l}
\hline
\textit{UnknownSalesperson} \\
sales, comm, bonus : 1 \mathrel{..} TeamSize \to \mathbb{N} \\
tot, ave, thresh : \mathbb{N} \\
ref? : \mathbb{N} \\
rep! : Report \\
\hline
tot = sumarray(sales) \\
ave = tot \ \mathbf{div}\ TeamSize \\
thresh = (2 * ave)\ \mathbf{div}\ 3 \\
ref? \notin 1 \mathrel{..} TeamSize \\
\hline
ref? \notin 1 \mathrel{..} TeamSize \\
rep! = \text{`}\textit{Unknown salesperson}\text{'} \\
\hline
\end{array}
$$

$$NotConfirmed = \{Basic\ Conversion\ Law\ 2\}$$

$$\begin{array}{|l}
\hline
\textit{NotConfirmed} \\
\hline
sales, comm, bonus : 1 \mathinner{..} TeamSize \to \mathbb{N} \\
tot, ave, thresh : \mathbb{N} \\
amnt?, conf? : \mathbb{N} \\
rep! : Report \\
\hline
tot = sumarray(sales) \\
ave = tot \ \mathbf{div} \ TeamSize \\
thresh = (2 * ave) \ \mathbf{div} \ 3 \\
amnt? \neq conf? \\
\hline
amnt? \neq conf? \\
rep! = \text{`}Input\ not\ confirmed\text{'} \\
\hline
\end{array}$$

Since all Z schemas can be converted into modified Z schemas using the basic conversion laws, the following laws give the conversion of schema conjunctions and schema disjunctions. Schema conjunctions can be converted into sequential compositions if the schemas involved have no common state variables.

Law 3 (Schema Conjunction). *If schema* $S1 \mathrel{\widehat{=}} [d1 \mid pre_1 \mid post_1]$ *and schema* $S2 \mathrel{\widehat{=}} [d2 \mid pre_2 \mid post_2]$ *and* $d1 \cap d2 = \varnothing$*, then*

$$S1 \wedge S2 \sqsubseteq S1;\ S2 \qquad \text{and} \qquad S1 \wedge S2 \sqsubseteq S2;\ S1$$

The schema *AddSalesOkay* can be refined to a sequential composition by using the *Schema Conjunction Law.*

$$\begin{array}{l}
AddSalesOkay \sqsubseteq \{Schema\ Conjunction\ Law\ 3\} \\
\quad AddSales;\ Success;
\end{array}$$

If the cases joined by schema disjunctions are mutually exclusive, we can prove that schema disjunctions can be refined by alternations.

Law 4 (Schema Disjunction). *If schema* $S1 \mathrel{\widehat{=}} [d1 \mid pre_1 \mid post1]$ *and schema* $S2 \mathrel{\widehat{=}} [d2 \mid pre_2 \mid post_2]$*, then*

$$S1 \vee S2 = \mathbf{if}\ \ (pre_1)S1;\ \mathbf{else\ if}\ (pre_2)S2;$$

Now we refine the schema operation *DoAddSales*. First calculate the preconditions of the following schemas,

$$\begin{array}{l}
\mathrm{pre}\ AddSalesOkay = ref? \in 1 \mathinner{..} TeamSize \wedge amnt? = conf?; \\
\mathrm{pre}\ UnknownSalesperson = ref? \notin 1 \mathinner{..} TeamSize; \\
\mathrm{pre}\ NotConfirmed = amnt? \neq conf?;
\end{array}$$

Since their conditions are exclusive, we can apply *Schema Disjunction Law* to the operation and get the following alternation:

$$\begin{array}{l}
DoAddSales = \{Schema\ Disjunction\ Law\ 4\} \\
\quad \mathbf{if}\ \ (ref? \in 1 \mathinner{..} TeamSize \wedge amnt? = conf?) \\
\qquad\qquad AddSalesOaky; \\
\quad \mathbf{else\ if}\ (ref? \notin 1 \mathinner{..} TeamSize) \\
\qquad\qquad UnknownSalesperson; \\
\quad \mathbf{else\ if}\ (amnt? \neq conf?) \\
\qquad\qquad NotConfirmed;
\end{array}$$

We also give the conversion law for schema compositions.

Law 5 (Schema Composition). *If schema operation* Op_1 *and* Op_2 *act over the same state, and they have no common output variables,* $(pre_C \wedge Op_1) \Rightarrow pre_2$*, where*

$$\begin{array}{l}
pre(Op_1 \mathbin{\text{\large ⨟}} Op_2) \mathrel{\widehat{=}} pre_C \wedge inv \wedge t \wedge t1 \wedge t2; \\
pre\ Op2 \mathrel{\widehat{=}} pre_2 \wedge inv \wedge t \wedge t2;
\end{array}$$

pre_C *is the precondition of* $Op_1;\ Op_2$*; inv is the state invariant; t,t1 and t2 are the restrictions that are introduced by the declarations of the state components, of the input variables of* $Op1$*, and of the input variables of* Op_2*, respectively, then*

$$Op_1 \mathbin{⨟} Op_2 \sqsubseteq Op_1;\ Op_2$$

3.2 Refinement Calculus: From Modified Z To C

The second step of system development is refining the modified Z specification to C code by applying appropriate refinement laws.

We developed a refinement calculus for modified Z, based on Morgan's refinement calculus that uses modified Z schemas instead of specification statements and develops the final code in C rather than the guarded command language.

The refinement calculi of Back, Morgan and Morris [1, 18, 17], are all based on the work of Dijkstra [8]. Each of the calculi creates a wide-spectrum language by extending a simple imperative programming language with specification constructs. A large number of refinement rules which capture traditional design intuitions and guarantee correctness have been proven using this characterisation. These rules range from simple laws which introduce local variables to complex laws which show that recursive procedures are correct. Our refinement rules are, on the whole, based on those of Morgan's calculus [17]. Based on the weakest precondition for modified Z [35] and C [32], all these rules have been proved in [34].

Law 6 (Alternation Introduction). *If* $pre \Rightarrow (\bigvee_{i=1..n} g_i)$ *and the predicate* g_i *has no free dashed variables and* $\cup_{i=1..n}\{g_i\} \neq \varnothing$*, then*

$$[d \mid pre \mid post] \sqsubseteq \textbf{if } (g_1)[d \mid g_1 \wedge pre \mid post];\ \cdots;\ \textbf{else if } (g_n)[d \mid g_n \wedge pre \mid post];$$

where $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$.

Law 7 (Assignment Introduction). *If* $pre \Rightarrow post[el/vl'][\alpha ds/\alpha ds']$*, where* dvl *declares the variables of* vl*,* vl *contains no duplicate variables,* vl *and* el *have the same length and have no free dashed variables, and the corresponding variables of* vl *and expression of* el *have the same type, then*

$$[d \mid pre \mid post] \sqsubseteq vl = el;$$

where $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$*,* $dvl \subseteq d$ *and* $dvl \cap di? = \varnothing$.

Obviously, the following assignment refines the schema *Success* by applying *Assignment Introduction Law.*

$$Success \sqsubseteq \{Assignment\ Introduction\ Law\ 7\}$$
$$rep! = \text{`}Okay\text{'};$$

Since it is not very easy to see the refinement result of the schema *AddSales*, we introduce some refinement laws to simplify the schema.

Law 8 (Weaken Precondition). *If* $pre \Rightarrow npre$*, then*

$$[d \mid pre \mid post] \sqsubseteq [d \mid npre \mid post]$$

Law 9 (Strengthen Postcondition). *If* $pre \wedge npost \Rightarrow post$*, then*

$$[d \mid pre \mid post] \sqsubseteq [d \mid pre \mid npost]$$

First, we apply *Weaken Precondition Law* to simplify the schema.

$$AddSales \sqsubseteq \{Weaken\ Precondition\ Law\ 8\}$$

AddSales

$sales, sales', comm, comm', bonus, bonus' : 1\,..\,TeamSize \to \mathbb{N}$
$tot, tot', ave, ave', thresh, thresh' : \mathbb{N}$
$ref?, amnt?, conf? : \mathbb{N}$

$tot = sumarray(sales)$

$tot' = sumarray(sales')$
$ave' = tot'\ \textbf{div}\ TeamSize$
$thresh' = (2 * ave')\ \textbf{div}\ 3$
$sales' = sales \oplus \{ref? \mapsto sales[ref?] + amnt?\}$
$tot' = tot + amnt?$
$comm' = comm$
$bonus' = bonus$

Since from

$$tot = sumarray(sales)$$
$$tot' = tot + amnt?$$
$$sales' = sales \oplus \{ref? \mapsto sales[ref?] + amnt?\},$$

we have

$$tot' = sumarray(sales').$$

So we can remove the above predicate from *AddSales* by applying *Strengthen Postcondition Law.*

$AddSales \sqsubseteq \{Strengthen\ Postcondition\ Law\ 9\}$

$$
\begin{array}{|l}
\hline
\textit{AddSales} \\
sales, sales', comm, comm', bonus, bonus' : 1 \mathrel{..} TeamSize \to \mathbb{N} \\
tot, tot', ave, ave', thresh, thresh' : \mathbb{N} \\
ref?, amnt?, conf? : \mathbb{N} \\
\hline
tot = sumarray(sales) \\
\hline
ave' = tot' \ \mathbf{div} \ TeamSize \\
thresh' = (2 * ave') \ \mathbf{div} \ 3 \\
sales' = sales \oplus \{ref? \mapsto sales[ref?] + amnt?\} \\
tot' = tot + amnt? \\
comm' = comm \\
bonus' = bonus \\
\hline
\end{array}
$$

The *Leading Assignment Law* is used to refine the above *AddSales* to separate assignments.

Law 10 (Leading Assignment Introduction). *If* $\{w\} \cap \{x\} = \varnothing$,

$$w, x = E, F[E/w] \equiv w = E;\ x = F;$$

By applying the above, we simplify *AddSales* as below.

$AddSales \sqsubseteq \{Leading\ Assignment\ Law\ 10\}$
$tot = tot + amnt?;$
$sales[ref?] = sales[ref?] + amnt?\};$
$ave = tot/TeamSize;$
$thresh = (2 * ave)/3;$
$comm = comm;$
$bonus = bonus;$

The *Skip Introduction Law* can be used to remove blank operations in the schema.

Law 11 (Skip Introduction). *If* $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$ *and* $pre \Rightarrow post[\alpha ds/\alpha ds']$, *then*

$$[d \mid pre \mid post] \sqsubseteq\ ;$$

And by applying *Skip Introduction Law*, we remove the last two assignments which are blank operations in *AddSales*.

$AddSales \sqsubseteq \{Skip\ Introduction\ Law\ 11\}$
$tot = tot + amnt?;$
$sales[ref?] = sales[ref?] + amnt?;$
$ave = tot/TeamSize;$
$thresh = (2 * ave)/3;$

The error schemas *UnknownSalesperson* and *NotConfirmed* can be simply refined to the following assignments individually:

$UnknownSalesperson \sqsubseteq \{Assignment\ Introduction\ Law\ 10\}$
$\quad rep! = `Unknown\ salesperson';$

$NotConfirmed \sqsubseteq \{Assignment\ Introduction\ Law\ 10\}$
$\quad rep! = `Input\ not\ confirmed';$

Now we get the following refinement result for *DoAddSales*:

```
if  (ref? ∈ 1 .. TeamSize ∧ amnt? = conf?)
{
      tot = tot + amnt?;
      ave = tot/TeamSize;
      thresh = (2 * ave)/3
      rep! = 'Okay';
}
else if (ref? ∉ 1 .. TeamSize)
      rep! = 'Unknown salesperson';
else if (amnt? ≠ conf?)
      rep! = 'Input not confirmed';
```

Fig. 1. The first refinement code of *DoAddSales*

The refinement code in Figure 1 is not executable, since it still contains some Z set operations and Z input/output operations. To obtain the final C code, we need to apply the following laws:

Law 12 (C Introduction). *We have the following rules for refining modified Z to final C code:*

1. *The Z* **div** *operation can be refined to the C / operation;*
2. *The input variables which have ? can be refined to the same input variables;*
3. *The output variables can be refined to C standard output statements:* **printf** *;*
4. *The Z set operations can be refined to corresponding C concrete data operations;*
5. *The overall schema operation can be refined to a C function with parameters which are the input variables of the schema operation.*

By applying the above, we obtain the final C code for *DoAddSales* in Figure 2.

The *Following Assignment Introduction Law* introduces an assignment following a schema.

Law 13 (Following Assignment Introduction). *If dvl declares the variables of vl, vl contains no duplicate variables, vl and el have the same length and have no free dashed variables, and the corresponding variables of vl and expression of el have the same type, then*

$$\begin{array}{l} \quad [d \mid pre \mid post] \\ \sqsubseteq [d \mid pre \mid post[el[ds', vl'/\alpha ds, vl]/vl'];\ vl = el; \end{array}$$

where $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$, $dvl \subseteq d$ *and* $dvl \cap di? = \varnothing$.

The **for** *Introduction Law* introduces **for** loop into specifications by refining a $\forall$ quantifier.

Law 14 (for Introduction). *If the specification P does not modify the variable i, then*

$$\forall\, i : I_{min} \mathrel{..} I_{max} \bullet P \sqsubseteq \{\mathbf{int}\ \ i;\ \mathbf{for}\ \ (i = I_{min};\ i <= I_{max};\ i++)P;\ \}$$

To illustrate the above laws, we introduce one more operation to the sales database.

```
void DoAddSales(int ref, int amnt, int conf)
{
      if ((1 <= ref <= TeamSize)&&(amnt == conf))
      {
            tot = tot + amnt;
            ave = tot/TeamSize;
            thresh = (2 * ave)/3
            printf ("Okay");
      }
      else if (!(1 <= ref <= TeamSize))
            printf ("Unknown salesperson");
      else if (amnt! = conf)
            printf ("Input not confirmed");
}
```

Fig. 2. The final code of *DoAddSales*

Example 3. *The schema Commission is used to calculate the sales commission.*

$$
\begin{array}{|l}
\textit{Commission} \\
\Delta \textit{SalesDB} \\
\hline
\forall\, i : 1 \mathrel{..} \textit{TeamSize} \bullet \\
\qquad \textit{comm}' = \textit{comm} \oplus \{ i \mapsto (\mathbf{if}\ \ \textit{sales}[i] > \textit{thresh} \\
\qquad\qquad \mathbf{then}\ (\textit{sales}[i] - \textit{thresh})\ \ \mathbf{div}\ \ 10\ \mathbf{else}\ 0)\} \\
\textit{sales}' = \textit{sales} \\
\textit{bonus}' = \textit{bonus} \\
\textit{tot}' = \textit{tot} \\
\textit{ave}' = \textit{ave} \\
\textit{thresh}' = \textit{thresh} \\
\hline
\end{array}
$$

Since the operation specified by Commission always succeeds, CommissionOkay and DoCommission are defined to be the same schema:

$$
\begin{aligned}
&\textit{CommissionOkay} \mathrel{\widehat{=}} \textit{Commission} \wedge \textit{Success}, \\
&\textit{DoCommission} \mathrel{\widehat{=}} \textit{CommissionOkay}.
\end{aligned}
$$

By applying *Following Assignment Introduction* to *Commission*, we get the following result:

$$\textit{Commission} \sqsubseteq \{\textit{Following Assignment Introduction Law } 13\}$$

$$
\begin{array}{|l}
\textit{Commission}1 \\
\Delta \textit{SalesDB} \\
\hline
\hline
\forall\, i : 1 \mathrel{..} \textit{TeamSize} \bullet \\
\quad \textit{comm}' = \textit{comm} \oplus \{ i \mapsto (\\
\qquad \mathbf{if}\ \ \textit{sales}[i] > \textit{thresh} \\
\qquad \mathbf{then}\ (\textit{sales}[i] - \textit{thresh})\ \ \mathbf{div}\ \ 10 \\
\qquad \mathbf{else}\ 0)\} \\
\hline
\end{array}
\quad ; \quad
\begin{array}{l}
\textit{sales}', \textit{bonus}', \textit{tot}', \textit{ave}', \textit{thresh}' \\
\quad = \textit{sales}, \textit{bonus}, \textit{tot}, \textit{ave}, \textit{thresh}
\end{array}
$$

The second part of the sequential composition can be refined to a **skip**, which is an empty statement in C, by the *Skip Introduction Law*. By applying the **for** *Introduction Law*, *Commission*1 is refined to a **for**-loop:

$Commission1 \sqsubseteq \{\textbf{for}\ \ Introduction\ Law\ 14\}$

```
  { int k;
    for (k = 1; k <= TeamSize; k++)
        if (sales[k] > thresh)comm'[i] = (sales[i] - thresh)/10;
        else comm'[i] = 0;
  }
```

$\sqsubseteq \{Assignment\ Introduction\}$

```
  { int k;
    for (k = 1; k <= TeamSize; k++)
        if (sales[k] > thresh)comm[i] = (sales[i] - thresh)/10;
        else comm[i] = 0;
  }
```

Similarly to *AddSalesOkay*, the schema *CommissionOkay* can be refined to a sequential composition by using the *Schema Conjunction Law.*

$CommissionOkay \sqsubseteq \{Schema\ Conjunction\ Law\ 3\}$
$\quad Commission;\ Success;$

The final code of *DoCommission* is obtained in Figure 3 by applying *C Introduction Law.*

```
void DoCommission(int ref, int amnt, int conf)
{
    {   int k;
        for (k = 1; k <= TeamSize; k++)
            if (sales[k] > thresh)
                comm[i] = (sales[i] - thresh)/10;
            else comm[i] = 0;
    }
    printf ("Okay");
}
```

Fig. 3. The final code of *DoCommission*

Our main objective is to fully formalise the refinement calculus for modified Z. Fortunately, a lot of work has been done based on specification statements and guarded commands [14, 4]. We present the rest of our refinement laws below.

Law 15 (Constant Introduction). *If* $pre \Rightarrow \exists\, cl : \textbf{cons} \bullet npre$*, where* $npre$ *has no free dashed variables, the constants* cl *and their dashed counterparts are not free in* $[d \mid pre \mid post]$*, then*

$$[d \mid pre \mid post] \sqsubseteq\ \{\textbf{cons}\ \ cl;\ [d \mid npre \mid post];\ \ \}$$

Law 16 (Constant Removal). *If the constants* cl *are not free in* p*, then*

$$\{\textbf{cons}\ \ cl;\ p;\ \} = p;$$

Law 17 (Contact Frame). *If the variables* $x, x' \notin d$*,*

$$[d;\ dx;\ dx' \mid pre \mid post] \sqsubseteq [d \mid pre \mid post[x/x']]$$

Law 18 (Expand Frame). *If the variables* $dx, dx' \notin \{d\}$*,* X *is fresh constant,*

$$[d \mid pre \mid post] \sqsubseteq\ \{\textbf{cons}\ \ X;\ [d;\ dx;\ dx' \mid pre \wedge x = X \mid post \wedge x' = X];\ \ \}$$

Law 19 (Fix Initial Value). *If the constants* $cl \mathrel{\widehat=} \cup_{i=1..n}\{c_i\}$, *the expression* e_i *has no free dashed variables, the corresponding constant of* c_i *and expression of* e_i *have the same type,* c_i *and* c_i' *are not free in* $[d \mid pre \mid post]$ *and in the corresponding expression of* e_i, *then*

$$[d \mid pre \mid post] = \{\mathbf{cons}\ \ cl;\ [d \mid pre \wedge (\textstyle\bigwedge_{i=1..n} c_i = e_i) \mid post];\ \ \}$$

Law 20 (Function Introduction - Call-by-value). *If* dx *declares parameter variables of function* $Fun()$, dx', *the corresponding dashed variables, function* $Fun()$ *is defined as below:*

$$\mathbf{void}\ \ Fun(\mathbf{ctype}\ \ x) \mathrel{\widehat=} [d;\ dx;\ dx' \mid pre \mid post],$$

with x' *not free in post, then*

$$[d \mid pre[A/x] \mid post[A/x]] \sqsubseteq Fun(A).$$

where $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$, *and* A *is values of the parameter variables.*

Law 21 (Function Introduction - Call-by-reference). *If* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$ *and* $\{d\} \cap \{d*x, d*x', d*a, d*a'\} = \varnothing$, *where* $d*x$ *declares pointer parameter variables of function* $Fun()$, $d*x'$, *the corresponding dashed pointer variables, and* $Fun()$ *is defined as below:*

$$\mathbf{void}\ \ Fun(\mathbf{ctype}\ \ *x) \mathrel{\widehat=} [d;\ d*x;\ d*x' \mid pre \mid post[*x'/*a']],$$

with $*x'$ *not free in pre, and neither* $*x$ *nor* $*x'$ *free in post, then*

$$[d;\ d*a;\ d*a' \mid pre \mid post] \sqsubseteq Fun(*a).$$

where $d*a$ *declares pointer variables* $*a$, $d*a'$, *the corresponding dashed pointer variables.*

Law 22 (Iteration Introduction). *If* vrt *is an integer, each* gi *and* vrt *have no free dashed variables, then*

$$\begin{array}{l} \quad [d \mid inv \mid inv[\alpha ds'/\alpha ds] \wedge \neg\, (\bigvee_{i=1..n} g_i[\alpha ds'/\alpha ds])] \\ \sqsubseteq \mathbf{while}\ \ (g_1 \mid\mid \cdots \mid\mid g_n) \\ \quad \{ \\ \quad\ \ \mathbf{if}\ \ (g_1)[d \mid inv \wedge g_1 \mid inv[\alpha ds'/\alpha ds] \wedge 0 \le vrt[\alpha ds'/\alpha ds] < vrt]; \\ \quad\ \ \cdots; \\ \quad\ \ \mathbf{else\ if}\ (g_n)[d \mid inv \wedge g_i \mid inv[\alpha ds'/\alpha ds] \wedge 0 \le vrt[\alpha ds'/\alpha ds] < vrt]; \\ \quad \} \end{array}$$

where $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$.

Law 23 (Sequential Composition Introduction). *If* mid *is a predicate and* cl *the constants,* x *the variables,* cl *and* x *have the same length and the constants of* cl *have the same type as the corresponding variables of* x, *the variables of* cl *and* cl' *are not free in* mid *and* $[d;\ dx;\ dx' \mid pre \mid post]$, *then*

$$\begin{array}{l} \quad [d;\ dx;\ dx' \mid pre \mid post] \\ \sqsubseteq \{\mathbf{cons}\ \ cl;\ [dx;\ dx';\ do1! \mid pre \mid mid];\ [d;\ dx;\ dx' \mid mid[cl/x][/'] \mid post[cl/x]];\ \ \} \end{array}$$

where dx *declares the variables of* x, dx', *the corresponding dashed variables,* $d \mathrel{\widehat=} ds \cup ds' \cup di? \cup do!$ *and* $do1! \subseteq do!$.

Law 24 (Variable Introduction). *If the variables of* vl *and* vl' *are not free in* $[d \mid pre \mid post]$ *and are not dashed, then*

$$[d \mid pre \mid post] = \ \{\mathbf{ctype}\ \ vl;\ [d;\ dvl;\ dvl' \mid pre \mid post];\ \}$$

where dvl *declares the variables of* vl.

Law 25 (Variable Renaming). *If the variables of* ul *and* ul' *are not free in* P *and* ul *are not dashed, then*

$$\{\mathbf{ctype}\ \ vl;\ P;\ \} \ = \ \{\mathbf{ctype}\ \ ul;\ P[ul, ul'/vl, vl'];\ \ \}$$

Law 26 (Variant Introduction). *If* pn *and* n *are not free in* e *and* $p1$, *then*

$$ps = \mid [\mathbf{procedure}\ pn \mathrel{\widehat=} (fpd \bullet \{n = e\}p1)\ \ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p2] \mid$$

4 Semantics

The semantics of both the programming constructs and the specification constructs of this language are given in terms of Dijkstra's weakest preconditions.

The correctness of program transformations is also characterised in terms of *weakest preconditions*, which were first introduced in [8]:

> The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given postcondition is called "the weakest precondition corresponding to that postcondition."

If the system is denoted by S and the desired postcondition by ψ, then we denote the corresponding weakest precondition by $wp.S.\psi$. If the initial state satisfies $wp.S.\psi$, it is guaranteed to establish eventually the truth of ψ. The meaning of $wp.S.\psi$ allows us to give the *wp* semantics for specification statements [17] and Z specifications [5]. This project also extends the *wp* semantics to modified Z specifications (Section 4.2) and C (Section 4.3).

Based on weakest preconditions, we give the formal definition for *refinement* ($\sqsubseteq$) between specifications(programs).

Definition 1 (Refinement). *For any specifications(programs) S_i and S_j,*

$$S_i \sqsubseteq S_j \ \textit{iff} \ \ \forall \psi, wp.S_i.\psi \Rightarrow wp.S_j.\psi.$$

The above allows the justification of the refinement laws introduced in Section 3(detail proof can be found in [34]) and also provides a basis for other laws to be derived.

4.1 *wp* Semantics for Z

In [5], the weakest precondition of a schema can be defined only if it specifies an operation, which can be written in the form $[ds;\ ds';\ di?;\ do! \mid p]$. Here ds declares the set of state variables, ds', the corresponding dashed variables, $di?$, the input variables, $do!$, the output variables and p is the predicate. The weakest precondition of the operation is

Definition 2 (Weakest precondition).

$$wp.[ds;\ ds';\ di?;\ do! \mid p].\psi \mathrel{\widehat{=}} (\exists\, ds';\ do! \bullet p) \wedge (\forall\, ds';\ do! \bullet p \Rightarrow \psi)$$

In the definition above, the operation is guaranteed to terminate exactly where there is a final state and outputs that satisfy p. Furthermore, it is guaranteed to establish the postcondition ψ upon termination if, whenever p holds, so does ψ.

4.2 *wp* Semantics for Modified Z

Definition 3 (Weakest precondition). *The weakest precondition of a modified Z schema $[d \mid pre \mid post]$ can be defined by*

$$\begin{aligned} & wp.[d \mid pre \mid post].\psi \\ \widehat{=}\ & pre \wedge (\forall\, ds', do! \bullet post \Rightarrow \psi) \end{aligned}$$

where d declares the set of all the schema variables: $d \mathrel{\widehat{=}} ds \cup ds' \cup di? \cup do!$. ds declares the set of state variables, ds', the corresponding dashed variables, $di?$, the input variables, $do!$, the output variables.

Schema Conjunction So we have the following definition:

Definition 4 (Schema Conjunction). *If schema* $S1 \mathrel{\widehat=} [d1 \mid pre1 \mid post1]$ *and schema* $S2 \mathrel{\widehat=} [d2 \mid pre2 \mid post2]$*, then*

$$S1 \wedge S2 \mathrel{\widehat=} \left[d1;\ d2 \left| \begin{array}{l} pre1 \wedge pre2 \\ \exists\, ds1';\ ds2';\ do1!;\ do2! \bullet post1 \wedge post2 \end{array} \right| post1 \wedge post2 \right]$$

Unfortunately, schema conjunction of modified Z is not monotonic with respect to the refinement relation. Mathematically, a function $f : D \rightarrow C$ is monotonic if

$$\forall\, x, y \in D, x < y \Rightarrow f(x) < f(y).$$

Applied to the refinement calculus, this becomes:

Definition 5 (Monotonicity). *A refinement relation* $f : D \rightarrow C$ *is monotonic, iff*

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y), \forall\, x, y \in D.$$

The following example demonstrates the non-monotonic nature of the refinement of schema conjunction.

Example 4. *If schema* $S1 \mathrel{\widehat=} [x : \mathbb{N} \mid true \mid x' = 1]$ *and schema* $S2 \mathrel{\widehat=} [x : \mathbb{N} \mid true \mid x' = 2]$*, then*

$$S1 \sqsubseteq x := 1 \qquad \text{and} \qquad S2 \sqsubseteq x := 2,$$

If we put the final code of $S1$ *and* $S2$ *together, the final program will be:*

$$x := 1 \wedge x := 2.$$

Though we get the final program, the program cannot itself be implemented since it is impossible to make the variable x *take two different values in the same state. In [19], such a program is called a* **miracle**.

But by the Definition 4, we have:

$$\begin{array}{l} \quad S1 \wedge S2 \\ = [x : \mathbb{N} \mid true \wedge (\exists\, x' \bullet x' = 1 \wedge x' = 2) \mid x' = 1 \wedge x' = 2] \\ = [x : \mathbb{N} \mid false \mid false] \end{array}$$

The program never guaranteed to terminate since the precondition is false. As a result, it allows any refinement at all: infinite loops, programs setting x *to arbitrary values, even programs that change other variables. In [17], it is called* **abort**. *So we get two different refinement results.*

Thus, if we have $S1 \wedge S2$, we cannot refine $S1$ or $S2$ individually. We must first remove these operations before we start the refinement. The techniques suggested consist of

1. using the *wp* definitions directly or
2. using refinement laws similar to King's rules [14]

to translate schema expressions. In [14, 25, 4], some transformation rules are given to convert Z schema conjunctions to sequential compositions in the refinement calculus.

Miraculous Schema Conjunction [25] introduces miraculous operations for specification statements, which build up an intuitive interpretation for the conjunction of two specification statements. The miraculous specification conjunction behaves like the specification conjunction, $\curlywedge$, when its result is feasible, but it does not guarantee a feasible result. Similar to specification statements, we give the definition of the miraculous schema conjunction for modified Z schema, $\textcircled{\scriptsize$\wedge$}$, which behaves similarly to Z schema conjunction.

Definition 6 (Miraculous Schema Conjunction). *If schema* $S1 \mathrel{\widehat=} [d1 \mid pre1 \mid post1]$ *and schema* $S2 \mathrel{\widehat=} [d2 \mid pre2 \mid post2]$*, then*

$$S1 \textcircled{\scriptsize$\wedge$} S2 \mathrel{\widehat=} [d1;\ d2 \mid pre1 \wedge pre2 \mid post1 \wedge post2]$$

From Definition 6, we can see that miraculous schema conjunction is monotonic with respect to the refinement relation, since we do not retain

$$\exists\, ds1';\ ds2';\ do1!;\ do2! \bullet post1 \wedge post2$$

in the definition.

Theorem 1 (Monotonicity of Miraculous Conjunction). *The miraculous schema conjunction is monotonic with respect to the refinement relation. That is, for modified Z schemas $S1$, $S2$, $S3$ and $S4$, if $S1 \sqsubseteq S3$ and $S2 \sqsubseteq S4$, such that*

$$S1 \circledwedge S2 \sqsubseteq S3 \circledwedge S4.$$

This means that each of its conjunctions can be refined before the operation is removed. However, since we remove the above formula in the definition, there is less information in the precondition of the conjunction. And there might be a problem because of the missing information. Take the following example:

Example 5. *Let S1 and S2 be the same as Example 4, then*

$$\begin{aligned} & S1 \circledwedge S2 \\ = & [x : \mathbb{N} \mid \mathit{true} \mid x' = 1 \wedge x' = 2] \\ = & [x : \mathbb{N} \mid \mathit{true} \mid \mathit{false}] \\ = & \ \mathbf{magic} \end{aligned}$$

No computer can execute that program, no contract based on it could be met. We called this **magic**.

In the above example, $S1 \circledwedge S2$ leads to the infeasible program, which always terminates and establishes the impossible **false**. So, in some cases, miraculous schema conjunction might not allow the Z-like incremental style, in spite of its monotonicity. Since we need to develop the program incrementally, as well as to use the monotonicity property of schema operations, it is important for us to check the feasibility of the operations before applying the refinement process. So we give the following definition:

Definition 7 (Feasibility). *The schema $S \mathrel{\widehat=} [d \mid pre \mid post]$ is feasible iff*

$$pre \Rightarrow (\exists\, ds', do! \bullet post),$$

where ds declares the state variables of S, ds', the dashed variables, $do!$, the output variables of S.

Based on the check for feasibility of the schema, we have the following theorem for the schema conjunction and the schema miraculous conjunction.

Theorem 2 (Schema Conjunction and Miraculous Conjunction). *If $S1 \wedge S2$ and $S1 \circledwedge S2$ are defined as above, then we have the following conclusions:*

1. $S1 \wedge S2 \sqsubseteq S1 \circledwedge S2$;
2. *If* $\exists\, ds1';\ ds2';\ do1!;\ do2! \bullet post1 \wedge post2$ *holds, then*
   $S1 \wedge S2 = S1 \circledwedge S2$;
   Else
   $S1 \wedge S2 = [d \mid \mathit{false} \mid post] =$ **abort**.

The first conclusion of the above theorem means, the miraculous schema conjunction always refines the schema conjunction, since it contains the weaker precondition.

The second conclusion says, if the two specifications are feasible, then the conjunction of them are also feasible. Given this condition, the schema conjunction of them is equal to the miraculous one. Thus, during refinement development, we can use the miraculous schema conjunction, which is monotonic with respect to the refinement relation, rather than the schema conjunction. Otherwise, at least one of two specifications is infeasible. So the precondition of the schema conjunction of them is false and the final program is equal to **abort**, since it can never be executed.

The following example demonstarates the use of miraculous schema conjunction in compositional refinement develoment.

Example 6.

$$
\begin{array}{l}
AddSalesOkay \sqsubseteq \{Schema\ Conjunction\ and\ Miraculous\ Conjunction\ Theorem\ 2\} \\
\quad AddSales \circledwedge Success; \\
\sqsubseteq \{Monotonicity\ of\ Miraculous\ Conjunction\ Theorem\ 1\} \\
\quad \{tot = tot + amnt?;\ sales[ref?] = sales[ref?] + amnt?;\ ave = tot/TeamSize;\ thresh = (2 * ave)/3;\ \} \\
\qquad\qquad \wedge \\
\quad rep! = \text{`}Okay';\\
\sqsubseteq \{Schema\ Conjunction\ Law\ 3\} \\
\quad tot = tot + amnt?; \\
\quad sales[ref?] = sales[ref?] + amnt?; \\
\quad ave = tot/TeamSize; \\
\quad thresh = (2 * ave)/3; \\
\quad rep! = \text{`}Okay';
\end{array}
$$

Schema Disjunction We have the following definition:

Definition 8 (Schema Disjunction). *If schema* $S1 \mathrel{\widehat{=}} [d1 \mid pre1 \mid post1]$ *and schema* $S2 \mathrel{\widehat{=}} [d2 \mid pre2 \mid post2]$*, then*

$$S1 \vee S2 \mathrel{\widehat{=}} \left[d1;\ d2 \left| \begin{array}{l} pre1 \vee \\ pre2 \end{array} \right| \begin{array}{l} (pre1 \wedge post1) \vee \\ (pre2 \wedge post2) \end{array} \right]$$

The feature that two cases joined by $\vee$ in Z specifications are mutually exclusive is as same as alternation. The following theorem shows that the schema disjunction can also be defined in terms of alternation if joined cases of the conjunction are mutually exclusive.

Theorem 3 (Schema Disjunction as Alternation). *If* $S1$*,* $S2$ *and* $S1 \vee S2$ *are defined as Definition 8,* $S1$ *and* $S2$ *are mutually exclusive, then*

$$S1 \vee S2 = \textbf{if}\ \ (pre1)S1;\ \textbf{else if}\ (pre2)S2;$$

The above theorem and the *Schema Disjunction Law* show that, during refinement, schema disjunction can be refined as an alternation.

4.3 *wp* Semantics for C

For the sake of brevity, we only present the semantics of C statements which we used in the sales database example. More detail can be found in [32].

Simple C Statements We define the semantics for simple C statements in terms of the weakest precondition with respect to a postcondition ψ.

$$
\begin{aligned}
wp.;\ .\psi &\mathrel{\widehat{=}} \psi \\
wp.\ \textbf{abort}();\ .\psi &\mathrel{\widehat{=}} \textbf{false} \\
wp.(x = E;\ ).\psi &\mathrel{\widehat{=}} \psi[E/x] \\
wp.(P;\ Q;\ ).\psi &\mathrel{\widehat{=}} wp.P.(wp.Q.\psi) \\
wp.\{\textbf{ctype}\ \ x;\ P;\ \}.\psi &\mathrel{\widehat{=}} \forall\, x : \textbf{ctype} \bullet wp.P.\psi \\
wp.\{\textbf{cons}\ \ c;\ P;\ \}.\psi &\mathrel{\widehat{=}} \exists\, c : \textbf{cons} \bullet wp.P.\psi
\end{aligned}
$$

The logical constants **cons** are fixed variables that take certain values such that the precondition holds. So they are different from the constants **const** in C, which must be defined as values at he very beginning of the program. Like variables, logical constants are declared. Since they have a different nature than variables, they are declared with the keyword **cons** . Unlike **ctype** , a logical constant declaration **cons** is not code, and hence at some later stage of refinement it must be removed. Naturally, that can occur only when all references to those logical constants have been eliminated, since otherwise they would become undeclared. And since logical constants are not code, specifying the type for them is optional.

Alternation Alternation is implemented by **if** – **else if** – $\cdots$ – **else if**. In general, this can be written as:

$$
\begin{array}{l}
\textbf{if}\ \ (G_1) \\
\qquad P_1; \\
\textbf{else if}\ (G_2) \\
\qquad P_2; \\
\vdots \\
\textbf{else if}\ (G_n) \\
\qquad P_n;
\end{array}
$$

Following the weakest precondition semantics for alternation in guarded command language in [17], we have the similar definition for C alternation:

$$wp.(\textbf{if}\ \ (G_1)P_1;\ \cdots\ \textbf{else if}\ (G_n)P_n;\ ).\psi \mathrel{\widehat{=}} (\bigvee_{i=1..n} G_i) \wedge (\bigwedge_{i=1..n} G_i \rightarrow wp.P_i.\psi)$$

Iteration First we give the semantics for C **while** loop, then derive the semantic for **for** loop. Following the guarded command language in [17], we give the semantics of three category of iterations separately.

If we write G as a Boolean expression of C and P is a segment of C code, then the **while**-loop can be expressed as:

$$
\begin{array}{l}
\textbf{while}\ \ (G) \\
\qquad P;
\end{array}
$$

We give the semantics for **while** loop below.

$$
\begin{array}{l}
\quad wp.\{\textbf{while}\ \ (G)P;\ \}.\psi \\
= \exists\, k \geq 0 \bullet H_k(\psi)
\end{array}
$$

The conditions $H_k(\psi)$ are given by

$$H_k(\psi) \mathrel{\widehat{=}} \begin{cases} \psi \wedge \neg\, G & k = 0, \\ wp.\{\textbf{if}\ \ (G)P;\ \}.H_{k-1}(\psi) \vee H_0(\psi) & k > 0. \end{cases}$$

We can derive the semantics for **for**-loop from the above definition. If we write a typical **for**-loop as:

$$
\begin{array}{ll}
\{ & \textbf{int}\ \ i; \\
 & \textbf{for}\ \ (i = I_{min};\ i <= I_{max};\ i++) \\
 & \qquad P; \\
\} &
\end{array}
$$

then we have the following derivation:

$$
\begin{array}{l}
\quad wp.\{\textbf{int}\ \ i;\ \textbf{for}\ \ (i = I_{min};\ i <= I_{max};\ i++)P;\ \}.\psi \\
= wp.(i := I_{min};\ \textbf{while}\ \ (i \leq I_{max})\{P;\ i := i+1;\ \}).\psi \\
= wp.i := I_{min}.(wp.\ \textbf{while}\ \ (i \leq I_{max})\{P;\ i := i+1;\ \}).\psi) \\
= wp.i := I_{min}.\exists\, k \geq 0 \bullet H_k(\psi) \\
= \exists\, k \geq 0 \bullet H_k(\psi)[I_{min}/i]
\end{array}
$$

The conditions $H_k(\psi)$ are given by

$$H_k(\psi) \mathrel{\widehat{=}} \begin{cases} \psi \wedge \neg\,(i <= I_{max}) & k = 0, \\ wp.\{\textbf{if}\ \ (i <= I_{max})\{P;\ i++;\ \}\}.H_{k-1}(\psi) \vee H_0(\psi) & k > 0. \end{cases}$$

Functions One of the most basic way of structured programming is the use of procedures, since it makes program more clear and concise. A procedure is defined in just one place but used in many.

In C, procedures are known as functions. Basically there exist two kind of C functions: functions without parameters and functions with parameters.

A function without parameters is extremely simple: a function is declared by associating a name with a program fragment, and the fragment is then called the *body* of the function.

void $Fun()\{\ P;\ \}$

In the above definition, *Fun* is the name of the function, and P is the function body, which is made up of C statements. The semantics of this kind of function is straightforward:

$$Fun() = P,$$
$$wp.Fun().\psi = wp.P.\psi.$$

Functions which have parameters are widely used in C. In addition to the benefits from structured programming, the use of this form of function makes the programmer more free to have the same code for different programs and to reuse the code.

Traditional functions with parameters have two different formats in C: $call-by-value$ and $call-by-reference$. For the sake of brevity, we only give the semantics for $call-by-value$ function below. The semantics for $call-by-reference$ function can be found in [35].

The $call-by-value$ function is the simplest parameterized function. The function is declared by associating a name with a defined variable and a program fragment, and the defined variable is then called the *parameter* of the function.

void $Fun($**ctype** $x)\{\ P;\ \}$

In the above definition, *Fun* is the name of the function, the parameter x has the type **ctype** , and P is the function body, which is made up of C statements.

A function call to *Fun* can be written as $Fun(a)$, where the variable a has compatible type with the parameter of *Fun*. The semantics of the function call can be given by:

$$Fun(a) = \{dl;\ l = a;\ P[l/x]\}.$$

In the above definition, dl declares the variable l which has the same type of the variable x. So we have the following derivation:

$$\begin{aligned} & wp.Fun(a).\psi \\ = {} & wp.\{dl;\ l = a;\ P[l/x]\}.\psi \\ = {} & \forall\, dl \bullet wp.\{l = a;\ P[l/x]\}.\psi && [wp \text{ semantics}] \\ = {} & (wp.P.\psi)[a/x] \end{aligned}$$

5 Conclusion

The novel contributions of this paper are as follows. This paper presents a formal software development method by integrating the Refinement Calculus with Z. To keep the incremental software development style of Z through the development process, modified Z [33] is introduced as the language of the refinement calculus, which supports the refinement of Z. A weakest precondition semantics has been given for modified Z [35] and standard C [32] as well. This allows the justification of the refinement rules introduced in the refinement calculus and also provides a basis for other laws to be derived. A refinement calculus for modified Z has been developed [31], along with the proof of all the refinement laws. Complete case study of the sales database can be found in [30]. Our objective is to develop a formal tool to support specification and refinement with Z. The conceptual design of a refinement tool prototype, ZRefiner, is proposed in the paper [33].

Several proposals for solving these problems can be found in the literature. The first proposal for integrating Z with the Refinement Calculus is presented in [14], where the differences between Z and the notation of the Refinement Calculus are analyzed and, in the light of these considerations, rules that translate schemas and some schema expressions to programs are suggested.

[27] has proposed a different form of integration, where schemas themselves have been regarded as commands of the language of the refinement calculus. Refinement of schemas is accomplished either by laws that are similar to the translation rules in [14] that apply to schema expressions, or by verification instead of calculation, when none of these laws apply. There is no equivalent to the rule of [14] that translates any schema.

Another approach is suggested in [25], where generalizations of the Z conjunction and disjunction schema operators are introduced into the language of the refinement calculus so that specification statements can be combined and the Z incremental style of building specifications can be used. The aim is to achieve a refinement calculus that can cope with large specifications. Moreover, a syntactic method of calculating Morgan's least conjunctive refinement operator for predicate tranformers is developed in [16]. Similar predicate transformer versions of the relational conjunction and disjunction operators are considered in detail. Then three fully-compositional operators, namely promotion envelop, simultance execution and standard alternation are given to support promotion in the refinement calculus. However the other schema operators, which also contribute to the success of the Z style, are not considered and it is not clear how they can be added to the refinement calculus. Our approach is similar to [25], but we propose to use Z throughout the whole development process, since it is more readable than specification statements.

[22] defined a notation for documenting the development of Ada programs from Z specifications. Despite the fact that a language similar to that of the refinement calculus was used, the proposal consisted of designing the programs directly in Ada and then giving an account of their correctness by using the notation and literate programming was suggested as part of Cleanroom, a method of software development that recommended the use of formal specifications and refinement, but that did not indicate any particular technique.

In [26], a refinement calculus for nondeterministic expressions is developed to transform highly abstract specifications into programs written in a functional programming language. The refinement calculus is built by adding specification constructs to the functional language and examining transformations over the resulting language.

The objectives of producing efficient programs instead of prototypes, and of applying a development method that has a mathematically sound basis and allows the use of calculational techniques are certainly best served by the approaches advocated in [14, 29, 27]. Nevertheless, they involve a conversion of notation that has not yet been formally justified.

In [4–6], ZRC - a refinement calculus for Z based on Morgan's calculus [17] was developed. The method allows specification using Z schemas and provides some conversion laws to convert Z schemas into the specification statements in the refinement calculus. Then the refinement is done in the refinement calculus. Since the schema operations are not monotonic with respect to the refinement relation, all these conversions should be applied during the early stage of the refinement. The drawback of this approach is that the abstraction of Z is lost as soon as the refinement starts.

In our integrated refinement calculus, we try to maintain an open-world view. We extend schema conjunction to be monotonic with respect to the refinement relation. Thus, we can leave the refinement of schema conjunctions to the later stage and keep the Z incremental style of developing specifications. Moreover, if the cases joined by schema disjunctions are mutually exclusive, we prove that schema disjunctions can also be defined by alternations.

5.1 Future Work

Since the system we produced here is only a prototype system, a lot of work has to be done to make it practical. We are working on more case studies for the prototype system and more practical work on the tool development.

Our future work must include full formalisation of the refinement laws for Z, along with data refinement.

Much work is to be done on strategies for refining Z specifications. Simplicity and efficiency should be considered during refinement steps. Based on this, the new refinement laws may be introduced to justify some steps.

The main objective of our future research is to develop a fully integrated tool for Z which supports refinement and good graphical user interfaces. So we need to investigat how to develop a formal tool with support for specification and refinement of Z. A conceptual design of such a refinement tool is proposed with a three-tier architecture, which makes the tool implementation much easier, more flexible and efficient. With the three-tier structure of ZRefiner, it is easier to develop and refine its components separately. We can develop the components of each module first, and then integrate them into the three-tier architecture. Since the technologies for syntax checking, type checking and typesetting are widely available, reusing these components should not be a problem. For example, Zeta [3] provides an open framework to edit, analyse and animate Z specifications and formalisms which are mapped to Z. After that, there will be a lot of work to be carried out to build a good refinement engine for ZRefiner. Again, we can make use of an existing theorem prover, for example Z/EVES [21].

After we adapt some existing tools to ZRefiner, then, further real-size case studies should be done to demonstrate the usefulness of this formal method.

Acknowledgments

Special thanks to Dr. Ana Cavalcanti for her generosity in sending her thesis and papers to us and also valuable discussions. Thanks also to Mark Saaltink for his help with Z/EVES.

References

1. R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica* **25**, pp. 593-624, 1988.
2. F. L. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Puakner and P. Pepper. The Munich Project CIP: Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science p.292, Springer-Verlag, 1987.
3. B. Robert and G. Wolfgang The Zeta System Developers Guide (DRAFT). Technische Universitt Berlin, 2000.
4. A. L. C. Cavalcanti. A Refinement Calculus for Z. PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, Technical Monograph PRG-123,1997.
5. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. Formal Aspects of Computing, 10(3): 267-289, 1998
6. A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for Z. The Computer Journal, 41(1):1-15,1998.
7. B. P. Collins, J.E. Nicholls and I. H. Sorensen. Introducing Formal Methods: the CICS Experience with Z. IBM Hursley and PRG Oxford University, 1987.
8. E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.
9. A. Diller. *Z: An Introduction to Formal Methods.* 2nd edition, John Wiley & Sons, 1994.
10. A. Gupta. Formal Hardware Verification Methods: A Survey. Journal of Formal Methods in System Design, volume 1, pp 151-238, 1992.
11. Ian Hayes. Specification Case Studies. Prentice Hall, second edition, 1993.
12. M. G. Hinchey, J. P. Bowen. Applications of Formal Methods. Prentice Hall, 1995.
13. C. B. Jones. Systematic Software Development using VDM. Prentice Hall, 2nd edition, 1990.
14. S. King. Z and the Refinement Calculus. VDM'90 VDM and Z - Formal Methods in Software Development, volume 428 of Lecture Notes in Computer Science, pages 164-188, Kiel-FRG, Spriner-Verlag, April 1990.
15. T. Kropf. Formal Hardware Verification: Methods and Systems in Comparison. Springer-Verlag, LNCS 1287, 1997.
16. B. P. Mahony. The Least Conjunctive Refinement and Promotion in the Refinement Calculus. Formal Aspects of Computing, 11: 75-105, 1999.
17. C. C. Morgan. Programming from Specifications. Prentice Hall, 2nd edition, 1994.
18. J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming 9, pp. 287-306, 1987.
19. C. C. Morgan, P. H. B. Gardiner, and K. A. Robinson. On the Refinement Calculus. Springer-Verlag, 1993.
20. H. A. Partsch. Specification and Transformation of Programs : a Formal Approach to Software Development. Springer-Verlag, 1990.
21. Mark Saaltink. The Z/EVES System. ORA Canada, In ZUM'97: The Z Formal Specification Notation (10th International Conference of Z Users, Reading, UK, April 1997, Proceedings)
22. C.T. Sennet. Demonstrating the compliance of Ada programs with Z specifications. In C.B. Jones, R.C. Shaw, and T. Denvir (eds), *5th Refinement Workshop*, Workshops in Computing, London, pp. 70-87, Springer Verlag, 1992.
23. J. M. Spivey. The Z Notation - A Reference Manual, 2nd edition. Prentice Hall, 1992.
24. Mark Utting. An Object-Oriented Refinement Calculus with Modular Reasoning. PhD thesis, University of New South Wales, Australia, 1994.
25. N. Ward. Adding Specification Constructors to the Refinement Calculus. In J. C. P. Woodcock and P. G. Larsen, editors, FME'93: Industrial-Strength Formal Methods, volume 670 of Lecture Notes in Computing Science, pages 652-670. Springer-Verlag, 1993.
26. N. Ward. A Refinement Calculus for Nondeterministic Expressions. PhD thesis, University of Queensland, 1994.
27. J. B. Wordsworth. Software Development with Z. Addison-Wesley Publishers, 1992.
28. J. B. Wordsworth. Software Engineering with B. Addison-Wesley Publishers, 1996.
29. Jim Woodcock and Jim Davis. Using Z : specification, refinement, and proof. Prentice Hall, 1996.
30. B. Wu, D.R.W. Holton and L. Lai. Case Study: A Sales Database. Technical Report TR-00-01. Dept of Computing, University of Bradford, Jan 2000.
31. B. Wu, D.R.W. Holton and L. Lai. The Refinement Laws for Modified Z. Technical Report TR-00-02. Dept of Computing, University of Bradford, May 2000.
32. B. Wu, D.R.W. Holton and L. Lai. A Weakest Precondition Semantics for C. Technical Report TR-00-03. Dept of Computing, University of Bradford, May 2000.
33. B. Wu, D.R.W. Holton and L. Lai. Towards a Mechanised Software Development Method. Proceedings of The Fifth International Workshop on Formal Methods for Industrial Critical Systems, Berlin, April 3-4, 2000.
34. B. Wu, D.R.W. Holton and L. Lai. The Refinement Laws for An Integrated Refinement Calculus. Technical Report TR-00-05. Dept of Computing, University of Bradford, June 2000.
35. B. Wu and L. Lai. A Weakest Precondition Semantics for Modified Z. Technical Report TR-99-03. Dept of Computing, University of Bradford, Aug 1999.