IJICT International Journal of Information & Communication Technology

ITRC

Volume 1- Number 2- May 2009 (pp.21-27)

ON THE CORRECTNESS OF A TRANSLATION MAP BETWEEN SPECIFICATIONS IN Z AND SETL2 PROTOTYPE

Behnaz Changizi
Coordination Languages Group
Center of Mathematics and Informatics
Amsterdam, The Netherlands
b.changizi@cwi.nl

Seyyed Hassan Mirian Hossinabadi
Computer Department
Sharif University of technology
Tehran, Iran
hmirian@sina.sharif.edu

Received: September 12, 2008- Accepted: May 3, 2009

***Abstract*— Formal specification as a precise description of software requirements plays an important role in the software development processes. It can be used as a measurement for validating the artifacts of almost all stages in the development process. Hence, making effort on validating the correctness of the formal specification against the requirements in the very early stages of development is of a high value. Extracting prototype from formal specification can be a kind of such a validation. In this article, we propose a translation set of rules for building executable prototypes written in SetL2 language from formal specification in Z formal language. Then, we investigate the correctness of the translation with help of some lemmas based on weakest precondition predicate transformer and refinement relationship.**

Keywords-Formal Specification, Prototyping, SetL2, Z, Weakest Precondition Predicate Transformer.

چکیده- توصیف صوری به عنوان روشی دقیق برای بیان نیازمندیهای نرم افزار نقش مهمی در فرایند توسعه نرم افزار به عهده دارد. توصیف صوری می تواند به عنوان معیاری جهت اعتبارسنجی محصولات تولیدشده تقریبا در همه مراحل توسعه نرم افزار به کار گرفته شود. بنابراین، تلاش برای سنجش اعتبار توصیف صوری در مراحل ابتدایی فرایند تولید، تلاشی مفید است. یکی از روش های انجام چنین اعتبارسنجی، استخراج نمونه از توصیف صوری است. در این مقاله، ما روشی برای ترجمه توصیف صوری بیان شده در زبان صوری Z به یک نمونه قابل اجرا در زبان برنامهسازی SetL2 ارایه می کنیم. در ادامه، درستی این ترجمه با استفاده از قضیه هایی مبتنی بر مبدل گزاره ضعیف ترین پیش شرط و نیز رابطه پالایش بررسی می شود.

I. Introduction

Starting a software development by writing its formal specification has undeniable advantages [1]. Required behavior of a software can be described by formal specifications precisely and unambiguously. On the other hand, formal specifications are typically complex products. They need to be verified to ensure they are sound, consistent and complete in regards to the user requirements [2]. Such a validation is necessary for formal development of software in the real world [3].

This paper presents a transformation of formal specification in *Z* [4] to programs written in SetL2 [5]. It also shows the correctness of this formal translation.

II. Contributions Compared with Other Approaches

Conversion of formal specifications written in *Z* and *VDM* [6] to programs in logical or functional languages has been reported in some literature. Some works translate model-based specifications to Prolog programs. The good point in this approach is that operations defined through relations can be directly implemented. Unfortunately, there is no algorithmic transformation of first order logic into Horn clauses [1]. On the other hand, the model-based specification structure usually has a similar form to a functional program because it contains abstract types and companion operations [1]. So, other researchers use functional languages as the target of the translation [1], [3], [7], [8], and [9]. None of these works present a coherent set of translation rules. They just offer a translation in an ad-hoc manner. Except [4] none of them make any effort to investigate the correctness of translation. Translation presented in [4] has a formal basis and it investigates the correctness of translation. It uses a functional language, for example Miranda, as prototyping language. This functional language can be substituted by a higher-level language like SetL2 to avoid unnecessary details in the obtained prototype and to provide the code which is easy to modify. Besides, the translation rules presented in [4] are still far from a suitable form for being employed in an automated translation process. A more suitable form for translation rules is for example a grammar-like structure that provides a formal basis which allows the combination of rules in a precise manner. In this paper, we use a very high level prototyping language called SetL2 for developing target programs. SetL2 [6] and its sibling languages, for example ProSet [8], are very good choices for prototyping purpose. They have been used for prototyping cases in several works [2] and [8].

A. Prototyping

As we mentioned above, research has been done on converting the formal specifications to functional languages- they are well known for making prototypes [7]. Trying to execute formal specifications results in some limitations on the specification. Reference [9] presents an algorithm to determine whether the specification is executable or not. It also offers some definitions for explicit predicates. Conversion of a non-executable specification to the executable form is not a trivial work. Here, we do not make effort on such a conversion. Prototype is not a comprehensive or final system. Furthermore, the goal of prototyping is to make it possible for user and developer to discuss system functionality and its capabilities. The prototype version of a system should be a miniature of the complete system [8].

Identify applicable sponsor/s here. If no sponsors, delete this text box. (sponsors)

III. Constructing Prototypes

Before explaining the translation process, it is necessary to provide some background information for the reader. As mentioned, not all predicates in a formal specification are directly executable. There are some examples that illustrate the complexity of extracting executable statements from a typical non-explicit predicate. In fact, such work can be very complicated. On the other hand, a study of the Z specifications collected in Specifications Case Studies edited by Ian Hayes [18] shows that a majority (94%) of the operation schemas are explicit or can be converted to the explicit form by small changes [9]. Even in a non-explicit schema, there exists useful information which is worth being captured. Due to the fact that a prototype is not supposed to be a total system, one can skip translation of the non-executable parts. Here they will be converted to text descriptions.

A. Some definitions

It is easy to see that the different types of variables of a typical schema can be categorized in four distinct groups: Inp?, as input variables, Out!, as output variables, St, as before state variables and St′as after state variables.

Def 2.1.1. IStInp. Suppose Variable is set of variables in a Z specification schema, IStInp, is set of input and before-state variables can be defined as:

$IStInp =_{def} \{v : Variable \mid v \in St \cup Inp?\}$ (1)

Def 2.1.2. FStOut. Suppose Variable is set of variables in a Z specification paragraph. FStOut, set of output and after-state variables can be defined as:$FStOut =_{def} \{v : Variable \mid v \in St' \cup Out!\}$ (2)

Def 2.1.3. Condition predicate. Condition predicate is a predicate whose variables are only in IStInp.

Def 2.1.4. Operation predicate. Operation predicate is a predicate which has at least one variable of FStOut form.

Def 2.1.5. Simple predicate. Let E1 and E2 be expressions,

inrel be an infix relational operator like $\in$ and prerel be a prefix relational operator like then a predicate in one of the E1 inrel E2 or prerel E1 is a simple predicate.

Def 2.1.6. Simple definition. Suppose v' is an afterstate variable and E is an expression with variables of IStInp type, then $v' = E$ is called a simple definition. It is also said that v' is defined by E. Var[E] is the set of variables of E and E can be evaluated if all of its members are defined. Then v' is evaluated by $v' := E$[9].

B. *Weakest precondition*

According to [17], the operation of a specification can be defined as below: "If the initial state satisfies the precondition then change only the variables listed in frame so that the resulting final state satisfies the postcondition."The meaning of a command is known if for any postcondition, preconditions which guarantee termination in a final state satisfying the postcondition is known.

For command *prog* and postcondition *A*, suppose *wp(prog, A)* be the weakest precondition sufficient to be sure *prog* will terminate in a state holding *A*. Therefore, *wp* is a predicate transformer, because it transforms the postcondition *A* into the weakest precondition *wp(prog,A)*. *B* precondition will guarantee termination of *prog* in a state holding *A* if *B* $\Rightarrow$ *wp(prog, A)*.

For instance, the weakest precondition of the assignment statement *x:=E* for postcondition *A* is that if variable *x* in expression is replaced by *E*, *A* still will be true [17].

$$wp(x := E) =_{def} A[x \backslash E] \quad (3)$$

C. *Refinement*

Refinement $\sqsubseteq$ is a relation between commands. Def 2.3.1 Refinement: For any command *prog1* and *prog2*, *prog1* is refined by *prog2*, *prog1* $\sqsubseteq$ *prog2*, if for all postcondition *A*, *wp(prog1, A)* $\Rightarrow$ *wp(prog2, A)* is hold [17].

D. *SetL2*

The SETL2 programming language is a very high level language based on the theory and notation of finite sets. It is evolved from SETL, developed at New York University by J. T. Schwartz. SETL2 adds to SETL a syntax and name scoping closer to more recent imperative languages, full block structure, and procedures as first class objects. SETL provides two basic aggregate data types: unordered sets, and sequences (the latter also called tuples). The elements of sets and tuples can be of any arbitrary type, including sets and tuples themselves. Maps are provided as sets of pairs (i.e., tuples of length 2) and can have arbitrary domain and range types. Primitive operations in SETL include set membership, union, intersection, and power set construction, among others. It also provides quantified boolean expressions constructed using the universal and existential quantifiers of first-order predicate logic.

SETL provides several iterators to produce a variety of loops over aggregate data structures. To have a quick look on the structure of SetL2 programs, here is a sample SetL2 line of code which prints all prime numbers from 2 to N [20].

prin t(n in 2..N — forall m in 2..n - 1 — n mod m > 0);

IV. TRANSLATION OF SPECIFICATIONS TO PROTOTYPE

As mentioned, predicates of a typical specification can be divided into two distinct groups of condition and operation. Because of this fact that condition predicates just include the input and before state variables, they cannot change the value of any variables and then not the program state as a result. In contrast, operation predicates can change the state of program. If a condition predicate is next to some operation predicate, it will play the role of precondition for them. However, the state of program will be changed just when the preconditions are satisfied.

Therefore, here such predicates will be translated into *if* statements in the target prototype. [9] presents some definitions and an algorithm for identifying that whether a given predicate is definitive and explicit or not. The following definition formalize our transformation rules. Def 3.1 *Map* function. Let Z_{Op} be the set of *Z* operators and S_{Op} be the set of SetL2 operators. We define Map : $Z_{Op} \rightarrow S_{Op}$ as in Table 3.

Def 3.2 *Report* function. Let S_Z be the set of schema predicates in Z language, S_S be the set of SetL2 statements and *NonDef* be a non-definitive operation predicate. *Report* function converts *NonDef* to a text comment in SetL2.

$$\text{Report} : \_{z} \rightarrow 7 \ \_{s} \quad (4)$$

$$\text{Report (NonDef)} =_{def} \text{print (NonDef);}$$

Def 3.3 *Trans* function: Let V_Z be the set of *Z* variables, V_S be the set of variables of SetL2 statements and *NonDef* be a non-definitive operation predicate. *Trans* function converts Z variables to the corresponding variables in SetL2.

$$\text{Trans} : V_Z \rightarrow V_S \quad (5)$$

$$\text{Trans}(v) =_{def} V$$

$$\text{Trans}(v') =_{def} V$$

$$\text{Trans}(v?) =_{def} V \text{ in}$$

$$\text{Trans}(v!) =_{def} V \text{ out}$$

Def 3.4 *[ ]*μ transformer. Let $\_{z}$ be the set of schema predicates in Z, $\_{s}$ be the set of SetL2 statements, *c* be a constant value of Z, *C* be the equivalent constant in SetL2, *v* be a variable in Z, *V* be the equivalent SetL2 variable, *inrel* be an infix relational operator and *prerel* be a prefix relational operator. []μ, a transformer which translates a translatable Z predicate to the equivalent statement in SetL2, is defined in 6. (A translatable predicate is an explicit and definitive predicate. It means that there is enough information on how to calculate results of that predicate). [9] presents a precise definition for such predicates and also an algorithm for determining it and calculating its results. Notice that in our translation, predicates detected as non-translatable will be converted to the text comment.

$[]^{\mu} : \Sigma_z \mapsto \Sigma_s$ (6)

$[s_1 inrel s_2]_{\mu} =_{def} [s_1]\_Map(inrel)[s_2]^{\prime}$

$[prerel s]_{\mu} =_{def} Map(prerel)[s]^{\prime}$

$[c]_{\mu} =_{def} C$

$[v]_{\mu} =_{def} Trans(v)$

Def 3.5 $[\ ]_T$ transformer. Let Σ_z be the set of schema predicates in Z, Σ_s be the set of SetL2 statements, *Gen* be a predicate, *Con* be a condition predicate, *IStInp* = *st* be a definitive operation predicate and *NonDef* be a non-definitive operation predicate. $[]_T : \Sigma_z \rightarrow \Sigma_s$, a transformer which translates a schema specification to SetL2 prototype, is defined as:

$[]_T : \Sigma_z \rightarrow \Sigma_s$ (7)

$[Con\ Gen]_T =_{def}$ if $[Con]_T$ then $[Gen]_T$ end;

$[IStInp = st\ Gen]_T =_{def} [IstInp]_T := [st]_T; [Gen]_T$

$[NonDef\ Gen]_T =_{def} Report\ (NonDef)[Gen]_T$

$[Con]_T =_{def}$ if $[Con]_T$ then pass; end;

$[IStInp = st]_T =_{def} IStInp := [st]_T$

$[NonDef]_T =_{def} Report\ (NonDef)$

$[A[B \backslash C]]_T =_{def} [A]_T\ [[B]]_T \backslash [C]_T]$

$[wp(P,A)]_T =_{def} wp\ ([P]_T, [A]_T)$

In optimization phase, generated *if* statements can be converted into *if* .. *then* ... *else* ... if needed. *pass;* statement does nothing in SetL2.

A. *Weakest precondition of Z and SetL2 statement*

In order to investigate the correctness of the proposed translation mapping, we are in need for a formal basis. Weakest precondition, as mentioned in Section 2.2 can be used for defining a formal semantic of programs. Actually, it is widely used in the program transformation filed. Table 1 shows the weakest precondition for some Z statements. In Table 2 the weakest precondition corresponding to some SetL2 programs are found.

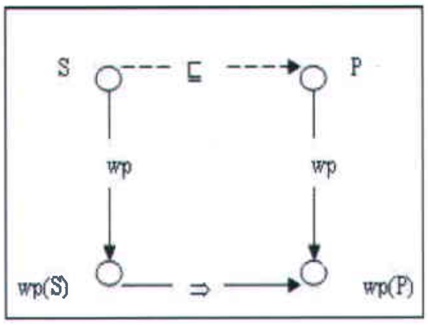

Fig 1. Refinement correlation & weakest precondition

V. PROOFS

It is clear that*Map* function maps the *Z* operators to equivalent operator in SetL2. It is easy to show that this map is semantically correct. It can be done by reasoning on the characteristics of each pair of this map. Lemma 1 investigates the correctness of map function for $\sharp$ operator. Here, to distinguish between two cardinality operators in Z and SetL2, we use $\sharp$ for Z and # for SetL2 cardinality operator.

Lemma 1. $\sharp$and # are semantically equivalent.

Proof: We use induction to show the equality. Assume that:

$A_i \sqsubseteq a_i$ (8)

$\sharp\varnothing = 0\ \#\{\} = 0$ [definition]

As the base of induction, we consider empty set.

$\sharp\{A_1,..,A_n\}=n,\ \#\{a_1,..,a_n\} = n$ [induction assumption]

Then we accept the correctness of above equations as assumption. And

$\sharp\{A_1, ..,A_n,A_{n+1}\} = \sharp\{A_1, ..,A_n\}+1 = n+1$

$\#\{a_1, ..,a_n,a_n\} = \#\{a_1, ..,a_n\}+1 = n+1$

Now we have proven the equality. Now using refinement co-relation, we show the correlation between the formal semantics of Z and SetL2 which has been defined by weakest precondition. Our main references for this section are [17] and [19]. Before starting proofs, we need to make some assumptions to build our proofs on top on them.

Assumption 1 $[A]_T = a$

a (of SetL2 domain) is the concrete form of A (of Z domain).

Assumption 2 $x = [IStInp]_T$

x (of SetL2 domain) is the concrete form of IStInp (of Z domain), while IStInp is an input or before state variable.

Assumption 3 $e = [E]_T$

e (of SetL2 domain) is the concrete form of E (of Z domain), while E is an expression.

Assumption 4 $p = [P]_T$

p (of SetL2 domain) is the concrete form of P (of Z domain), while P is an expression.

Assumption 5 $q = [Q]_T$

q (of SetL2 domain) is the concrete form of Q (of Z domain), while Q is an expression.

Assumption 6 $cond = [Cond]_T$

cond (of SetL2 domain) is the concrete form of Cond (of Z domain), while Cond is a condition predicate expression.

Assumption 7 $oper = [Oper]_T$

oper (of SetL2 domain) is the concrete form of Oper (of Z domain), while Oper is an operation predicate expression.

Assumption 8 $s = [S]_{\tau}$

s (of SetL2 domain) is the concrete form of S (of Z domain), while S is a set.

Assumption 9 $p(x) = [P(X)]_{\tau}$

p(x) (of SetL2 domain) is the concrete form of P(X) (of Z domain), while P(X) is a predicate on X.

Lemma 2. Generated statements using *[ ]*$_{\tau}$ are semantically correct. If *S* is a chain of schema predicates, then *[S]*$_{\tau}$ will be hold.

Proof. Using weakest pre-conditions defined in Table 1 and Table 2 and refinement relationship, we do induction on the Z expressions. First we show the correctness of refinement relation for basic structures. And we show that $IStInp = E \sqsubseteq x := a$.

The approach for proving this lemma is shown in [4]. So it means to show that S specification is refined by P program, it is enough to show that the wp(S) => wp(P). On the other hand, P actually is $[S]\tau$ (the transformation of S using τ function). Therefore, we need to show that wp(S) => wp($[S]\tau$). First, We have show the correctness of this correlation for single statements. Now we want to consider the combination of statements. We will start with the combination of two statements. Then using induction, we investigate k+1 statements elements.

First we apply the weakest precondition transformer on both of IStInp = E and x := a.

$wp(IStInp = E,A) = A[IStInp\backslash E]$ [Table 1]

$wp(x := e,a) = a[x \backslash e]$ [Table 2]

Now we apply τ operator on the result.

$[A[IStInp\backslash E]]_{\tau} = [A]_{\tau}[[IStInp]_{\tau}\backslash[E]_{\tau}] =_{def} a[x \backslash e]$ [Assumptions 1,2,3]

$wp(IStInp = E,A) = wp(x := a)$

And finally: $IStInp = E \sqsubseteq x := a$

We use same reasoning on PQ and pq.

$wp(PQ,A) = wp(P,wp(Q,A))$ [Table 1]

$wp(pq,a) = wp(p,wp(q,a))$ [Table 2]

$[wp(P,wp(Q,A))]_{\tau} = wp([P]_{\tau}, [wp(Q,A)]_{\tau}) =$

$wp(p,wp([Q]_{\tau}), [A]_{\tau})$ [Assumptions 1,4,5]

$[wp(P,wp(Q,A))]_{\tau} = wp(p,wp(q,a))]$

$PQ \sqsubseteq pq$

Now same with Cond Oper and if statement.

$wp(Cond\ Oper ,A) = Cond \wedge (Cond \Rightarrow wp(Oper ,A))$ [Table1]

wp(if cond then oper end,a) = cond and (cond impl wp(oper ,a)) [Table 2]

$wp(Cond\ Oper ,A) = [A]_{\tau}[[IStInp]_{\tau}\backslash[E]_{\tau}] =_{def}$

$a[x \backslash e]$ [Assumptions 1,6,7]

wp(IStInp = E,A) = wp(if cond then oper end,a)

Cond Oper $\sqsubseteq$ if cond then oper end

And with forall quantifier.

$wp(\forall X : S \mid P(X),A) = \forall X : S \mid wp(P(X),A)$ [Table1]

wp(forall x in s | p(x),a) = forall x in s | wp(p(x),a) [Table2]

$[\forall X : S \mid wp(P(X),A)]_{\tau}$ = forall x in s | wp(p(x),a) [Assumptions1,2,8,9]

$wp(\forall X : S \mid P(X),A)$ = wp(forall x in s | p(x),a)

$\forall X : S \mid P(X) \sqsubseteq$ forall x in s | p(x)

For existential quantifier.

$wp(\exists X : S \mid P(X),A) = \exists X : S \mid wp(P(X),A)$ [Table 1]

wp(exists x in s | p(x),a) = exists x in s | wp(p(x),a) [Table 2]

$[\exists X : S \mid wp(P(X),A)]_{\tau}$ = exists x in s | wp(p(x),a) [Assumptions 1,2,8,9]

$wp(\exists X : S \mid P(X),A)$ = wp(exists x in s | p(x),a)

$\exists X : S \mid P(X) \sqsubseteq$ exists x in s | p(x)

For unique existential quantifier.

$wp(\exists_1 X : S \mid P(X),A) = \exists_1 X : S \mid wp(P(X),A)$ [Table 1]

wp(justexistsone x in s | p(x),a) = justexistsone x in s | wp(p(x),a) [Table 2]

$[\exists_1 X : S \mid wp(P(X),A)]_{\tau}$ = justexistsone x $\in$ s | wp(p(x),a) [Assumptions 1,2,8,9]

$wp(\exists_1 X : S \mid P(X),A)$ = wp(justexistsone x in s | p(x),a)

$\exists_1 X : S \mid P(X) \sqsubseteq$ justexistsone x in s | p(x)

For non existential quantifier.

$wp(\nexists X : S \mid P(X),A) = \nexists X : S \mid wp(P(X),A)$ [Table 1]

wp(notexists x in s | p(x),a) = notexists x in s | wp(p(x),a) [Table2]

$[\nexists X : S \mid wp(P(X),A)]_{\tau}$ = notexistsone x in s | wp(p(x),a) [Assumptions 1,2,8,9]

$wp(\nexists X : S \mid P(X),A)$ = wp(notexists x in s | p(x),a)

$\nexists X : S \mid P(X) \sqsubseteq$ notexists x in s | p(x)

For induction, we assume that refinement relationship is true for a combination of *k* predicates of Z which are from different types. Then, here we show that by adding another predicates, relationships still hold. In the other words, we are going to prove this relation:

$Z_1Z_2...Z_n \sqsubset S_1S_2...S_n \Rightarrow Z_1Z_2...Z_nZ_{n+1} \sqsubset S_1S_2...S_nS_{n+1}$

The *k+1*th predicate can have any on possible forms. So it is necessary to consider all of the possible case. However, here we just show some of these different forms to show the flavor of proofs. The proof for the other forms will not be much different.

Case 1- Z_{n+1} is in *IStInp = E* form:
Proof:
$wp(Z_1Z_2...Z_n\ IStInp = E,A) = wp(Z_1Z_2...Z_n, wp(IStInp = E,A))$ [By definition]
$wp(Z_1Z_2...Z_n,A) \Rightarrow wp(S_1S_2...S_n, [A]_T)$ [Assumptions]
$wp(IStInp = E,A) \Rightarrow wp([IStInp]_T := [E]_T, [A]_T)$
[Shown earlier]
$Z_1Z_2...Z_n IStInp = E \sqsubseteq S_1S_2...S_n [IStInp]_T := [E]_T$

Case 2- Z_{n+1} is in *Cond* form:
Proof:
$wp(Z_1Z_2...Z_n,A) = wp(S_1S_2...S_n,A)$ [Assumptions]
$wp(Cond\ Z_1Z_2...Z_n,A) \Rightarrow wp(\text{if } [Cond] \text{ then } S_1S_2...S_n \text{ end}, [A]_T)$ [Shown earlier]
$Cond\ Z_1Z_2...Z_n \Rightarrow \text{if } [Cond]_T \text{ then } S_1S_2...S_n \text{ end}$

Case 3- Z_{n+1} is in $\forall X : S \mid P(X)$ form:
Proof:
$wp(Z_1Z_2...Z_n \forall X : S \mid P(X),A) = wp(Z_1Z_2...Z_n, wp(\forall X : S \mid P(X),A))$ [By definition]
$wp(Z_1Z_2...Z_n,A) \Rightarrow wp(S_1S_2...S_n, [A]_T)$ [Assumptions]
$wp(\forall X : S \mid P(X),A) \Rightarrow wp(\text{ forall } [X]_T \text{ in } [S]_T \mid [P(X)]_T, [A]_T)$ [Shown earlier]
$Z_1Z_2...Z_n \forall X : S \mid P(X) \sqsubseteq S_1S_2...S_n \text{ forall } [X]_T \text{ in } [S]_T \mid [P(X)]_T$
Deduction about other forms of quantifiers is also similar.

VI. Example

Now in an example, we show how *[ ]*$_T$ transformer can be applied on Z specifications. We just want to show the manner, so we use a simple case. 5 shows a simple schema in Z.We just go through this simple example to show the flavor of translation. Nonetheless, translation rule can be applied on much more complicated cases. *Update* schema specifies an update operation on *st*. After the operation, the pair of *s?, v?* should be added to *st*. Here,We apply [ ]T function on the top of statement.Then we will go further and substitute each part by its translation result step by step.

Update
$st, st' : sym \twoheadrightarrow val$
$s? : sym$
$v? : val$

$st' = st \cup s? \mapsto v?$

Fig. 2: Update Operation Schema

$[st' = st \cup s? \mapsto v?]_T \Rightarrow$
$[st']_T := [st \cup s? \mapsto v?]_T; \Rightarrow$
$st := [st]_T[\cup]_T[s? \mapsto v?]_T; \Rightarrow$
$st := st \text{ with } [s? \mapsto v?]_T; \Rightarrow$
$st := st \text{ with } [s?]_T[\mapsto]_T[v?]_T; \Rightarrow$
$st := st \text{ with } [s_{in}, v_{in}];$

So the result of translation will also imply that after its execution, **st** will be the union of **st** and the pair of $[s_{in}, v_{in}]$.

VII. Conclusion

This article describes a systematic way for prototyping Z specifications. A systematic method for prototyping helps to have fast and easy to modify prototype programs which is the main concern in prototyping. For investigating the correctness of the translation, we need to provide source and target of translation with a formal semantic. A formal semantic has been defined using weakest precondition which is a widely used theory in the program refinement field. In the other hand, refinement correlation has been used to make a correlation between these formal semantics of source and target. As future work, we are going to apply the translation rules on a case study and gather some statistical data over the correctness of translation in practice.

Table 1: Predicate transformers of SetL2 used for prototyping

$wp(IStInp = E,A) =_{def} A[IStInp \backslash E]$
$wp(Oper1Oper2,A) =_{def} wp(Oper1, wp(Oper2,A))$
$wp(Cond\ Oper,A) =_{def} Cond \wedge Cond \Rightarrow wp(Oper,A)$
$wp(\forall x : S \mid P(x),A) =_{def} \forall x : S \mid wp(P(x),A)$
$wp(\exists x : S \mid P(x),A) =_{def} \exists x : S \mid wp(P(x),A)$
$wp(\exists 1\ x : S \mid P(x),A) =_{def} \exists 1\ x : S \mid wp(P(x),A)$
$wp(\nexists x : S \mid P(x),A) =_{def} A$

Table 2: Predicate transformers of SetL2 used for prototyping

wp(skip,A) =def A
wp(x := E,A) =def A[x \E]
wp(PQ,A) =def wp(P,wp(Q,A))
wp(if G then P end,A) =def G and G impl wp(P,A)
wp(forall x in S | P(x),A) =def forall x in S | wp(P(x),A)
wp(exists x in S | P(x),A) =def exists x in S | wp(P(x),A)
wp(justexistsone x in S | P(x),A) =def justexistsone x in S | wp(P(x),A)
wp(notexistsxinS | P(x),A) =def A

REFERENCES

[1] J. M. Spivey, "The Z Notation: A Reference Manual", Third Edition, US, Prentice Hall, 2001.

[2] W. Hasselbring, "Prototyping Parallel Algorithms in a Set-Oriented Language", Hamburg, Kovac, 1994.

[3] C. B. Jones, "Systematic Software Development Using VDM", Prentice Hall International Series in Computer Science, 1990.

[4] A. E. Abdallah, A. Barros ,J. B.Barros, J. P Bowen., "Deriving Correct Prototypes from Formal Z Specifications", Technical Report SBU-CISM-00-27, SCISM, South Bank University, London, UK, 2000.

[5] T. Tilley, "Formal Concept Analysis Applications to Requirements Engineering and Design", Ph.D. Thesis, the University of Queensland, Australia, 2003.

[6] W. K. Synder, "The SetL2 Programming Language", Technical Report 490, Courant Institute, New York University, New York, 1990.

[7] P. Borba ,S. Meira, "A System for Translating Executable VDM Specifications into Lazy ML", Software - Practice and Experience, Vol. 27, No 3, pp. 271-289, 1997.

[8] E. E. Doberkat, W. Frank, U. Gutenbeil, W., U. Lammers, C. Pahl, "ProSet A Language for Prototyping with Sets", 3rd InternationalWorkshop on Rapid System Prototyping, IEEE Computer Society Press, Research Triangle Park, NC, pp. 235–248., 1992.

[9] X. Jia, "An Approach to Animating Z Specifications", 19th Annual IEEE International Computer Software and Applications Conference (COMPSAC'95), Dallas, Texas, USA, pp. 108- 113, 1995.

[10] P. Malik, M. Utting, "CZT: A Framework for Z Tools", 4th International Conference of B and Z Users (ZB'05), Springer Berlin / Heidelberg, Vol. 3455/2005, pp. 65-84, 2005.

[11] G. O'Neil, "Automatic Translation of VDM Specifications into Standard ML Programs", the Computer Journal, Vol. 35, No 6, 1992.

[12] B. Paulo, R. Silvio, "From VDM Specifications to Functional Prototypes", Journal of Systems and Software, Vol. 21, No 3, pp. 267-278, 1993.

[13] T. Miller, Paul Strooper, "A Case Study in Specification and Implementation Testing", 11th Asia-Pacific Software Engineering Conference, pp 130-139, IEEE Computer Society, 2004.

[14] P. T. Breuer, J. P Bowen., "Towards correct executable semantics for Z", Z User Workshop, Cambridge, Workshops in Computing, pp. 185 – 209, Springer-Verlag, 1994.

[15] C. Browne, Computer Languages, http://cbbrowne.com/info/functional.html

[16] M. Utting, P. Malik, Community Z Tools (CZT) project, http://czt.sourceforge.net

[17] M. Carroll, "Programming from Specifications", Second Edition, Prentice Hall, 1994.

[18] I.J Hayes., "Specification Case Studies", Second edition, London: Prentice-Hall, 1992.

[19] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, "Automated update management for XML integrity constraints", Workshop on Programming Language for XML (PLAN-X), 2002.

[20] http://en.wikipedia.org/wiki/SETL2

Behnaz Changizi received the B.Sc. degree in Software Engineering from Amir Kabir University of Technology (Tehran Polytechnic), Tehran, Iran in 2003, and the M.Sc. degree also in Software Engineering from Sharif University of Technology, Tehran, Iran in 2007. She started her Ph.D. program in 2008 in Leiden University, The Netherlands. Her current research interests include application of Formal Methods in software specification and software development, Verification, Model Transformation and Domain Specific Language. Her email address is b.changizi@cwi.nl.

Seyed-Hassan Mirian-Hosseinabadi received the B.Sc. degree in Software Engineering from Shahid Beheshti University, Tehran, Iran in 1984, and the M.Sc. degree also in Software Engineering from Sharif University of Technology, Tehran, Iran in 1987. He started his Ph.D. program in 1993 and received the Ph.D. degree in Computer Science (Formal Methods) from the University of Essex, Colchester, UK in 1996. He joined Sharif University of Technology in 1996, and is currently an Assistant Professor in
the Department of Computer Engineering. His current research interests include application of Formal Methods in software specification and software development in particular with the help of the Type Theory and Constructive Mathematics, software metrics and measurement, reconfigurable software architecture, formal specification and verification of software architecture, and Relational and XML databases. His email address is hmirian@sharif.edu.