Australian Journal of Basic and Applied Sciences, 6(8): 421-430, 2012
ISSN 1991-8178

# Algebraic Semantics of Concepts for A Simple Programming Language

Bashar Ziad Abu Sa and Abdullah Mohd Zin

Center for Software Technology and Management, Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia.

**Abstract:** Studying the semantics of programming languages is very important. New constructs are regularly added to programming languages that need to be described by using a formal semantics method. Concept is a new constructs that is suggested to C++ as a solution to some problems that appear in generic programming using C++ templates. This paper attempts to describe an algebraic semantics of concepts. In order to define the semantics, we first describe a mini-language called CDL, which is a small subset of C++. Our approach for defining the semantics is done by using parameterized specification to describe the transformation of concepts, concept maps and template classes.

**Key words:** Concepts, Algebraic Semantics, Generic Programming, Template Classes.

## INTRODUCTION

Concepts (Gregor, D., 2006) are constructs that were introduced as a tool for constraining the generic parameters in the template definition. The concept is an encapsulation of a set of requirements that must be fulfilled in order for an actual type to model that concept. This construct is a representation of the requirements of a generic type the programmer must be aware of when using the generic algorithms implemented by the C++ templates.

Generic programming is a programming technique that is used for the development of generic software libraries (Austern, M.H., 1998). In C++ (Stroustrup, B., 2000), templates are used to write generic algorithms due to its ability to define generic type parameters (Stroustrup, B., 1994). Nevertheless, using templates has a problem. The type parameters of templates are unconstrained data types, so, any type can be substituted for these generic types. Despite the benefit of the generic nature of template parameters in implementing generic algorithms, it includes a problem. The generic algorithm may assume some requirements on the types that it can be used with and if it is not satisfied by the actual type that is substituted in place of the generic parameter it will lead to an error that is not discovered at compilation time but the detection may be late at execution time. type checking is not performed until they are actualized with a specific data type. As solution to this problem, the concept construct was introduces as a suggested construct to be used in C++ in order to explicitly express the requirements of the implemented generic algorithm on its generic type parameters. Concepts are used to constrain the type parameters in C++ templates. Using concepts, templates can be type-checked at compilation time which leads to an early detection of the unfulfilled requirements of the generic algorithms at compilation time.

In this paper, we present a formal semantics model that represents simple concepts, concept maps and constraint template classes for a mini language that we define and call CDL. To simplify the problem, we defined CDL to have the same syntax of concepts, concept maps and template classes for that of C++ but with a single parameter. Our approach is based on giving the rules to transform each construct into an algebraic specification. For concepts and constrained template classes, we use the parameterized specification. For concept maps, we use the simple specification. Then, we give the rules to interpret the specification of the concept map and the actualization of the template class using algebras. For concepts we use the flat algebraic specification. Some properties related to concept that this model satisfy are presented and proved. For example, we proved that the concept map specification preserves the semantics of the actual parameter. We also proved that when the parameterized class is transformed after being type-checked using concept modeling type checking that it preserves the semantics of its parameter operations and the semantics of the operation of the constraining concept.

This paper is divided into 5 sections. In section 2 we give the related work. In section 3 we define the CDL language. In section 4 we give the semantics of CDL. In section 5 we give a conclusion and further work.

### *Related Work:*

Formal semantics of programming languages were studied by several researchers (Florent, K., 2007; Harmen, K., 2006; Mosses, P.D., 2006; Broy, M. and Wirsing, M., 1982; Broy. M., 1987). Algebraic semantics were used to study the semantics of functional (Broy, M. and Wirsing, M., 1982), imperative (Broy. M., 1987) and object oriented languages (Harmen, K., 2006).

**Corresponding Author:** Bashar Ziad Abu Sa, Center for Software Technology and Management, Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia.

In (Fronk, A., 2002), Fronk developed an algebraic semantics of a sample object oriented language. The approach encompasses two steps: First, developing an algebraic semantics of the basic object oriented concepts. Second, embedding the semantics of the imperative parts of the language.

Generic Programming is a systematic technique for software design and organization. It aims is to find the most general formulation of an algorithm with an efficient way of implementation as possible. The main objective of generic programming is to make algorithms as widely applicable as possible. Generic programming is sometimes referred to as algorithm-oriented (Jazayeri, M., 1998). In C++ language (Stroustrup, B. 1994; 2000), generic programming is supported through templates (Austern, M.H., 1998).

In (Gregor, D., 2006) Concept is a new construct that was suggested as an approach for type checking of templates in C++. This addition of concepts provided a methodology for modular type checking for templates in order to support generic programming by making template classes and methods easier to design and implement. This feature was suggested to be used in type checking and in compiler optimization (Tang, X. and Jarvi, J. 2007). Semantics analysis in compiler design can be found in (Wilhelm, R., 1995) and (Aho. A.V., 1996). For more details on programming languages and its semantics review (Pratt, T.W., 2001) and (Slonneger, K., 1995).

This new important language feature (like all other features of any programming language) to be designed, implemented and used correctly, it must be semantically described correctly and accurately. Using informal description of the semantics of concepts is not enough. It must be described formally to be easily used, designed and implemented.

***The Language CDL:***

The sublanguage CDL is a small subset of C++ that contains constructs for the purpose of defining the semantics of concepts. CDL contains concepts, concept maps and template class constrained by a concept but only with a single parameter. The main program is defined the same that is for C++:

***Concepts:***

A concept is a construct that describes a constraint on a type parameter. A concept in CDL is defined as follows:

concept ConceptName<typename TypePar>
{a group of operations}

where the operation is defined as follows:

mt mv(t1 p1, . . ., tn pn)

***Concept Maps:***

A concept map is a construct that is defined to state that a certain type models a certain concept and to describe how it models it. A CDL concept map is defined as follows:

concept_map ConceptName<TypeName>
{A group of operations}

where, each operation in the concept map is an implementation for the corresponding operation in the concept definition.

***Template Classes Constrained by Concepts:***

Template class defines a generic type parameter and can give a constraint on the nature of that parameter using a concept. The template class is defined as follows:

template <typename TypePar> where ConceptName<TypePar>

class ClassName
{Group of attributes Group of operations}

***An Example Program:***

We give an example program to clarify the CDL language concepts.

concept LessThanComparable<typename T>
{bool LessThan(T a,T b);}

```
class Complex {float RP; float IP; public: Complex(float r, float i) {RP=r; IP=i; }

Complex() {RP=0; IP=0;}

void SetRP(float r) { RP=r;}

void SetIP(float i) { IP=i;}

float getRP() {return (RP);}

float getIP(){return(IP);}

float abs(){ return(sqrt(RP*RP+IP*IP));}
};

concept_map LessThanComparable<Complex>
{bool LessThan(Complex a, Complex b)
    {
      return(abs(a)<abs(b));
    }
}

template <typename T> where LessThanComparable<T>
class ArithmeticOperations {
    T  min(T u, T v)
    {
        return(LessThan(u,v)?u:v);
    }
}

int main()
{
    Complex  x(2,3),y(-1,1),z;
    ArithmeticOperations <Complex> ArOp;
    Z= ArOp.min(x,y);
    cout<<"The minimum is: "<<z.getRP()<<
        "+"<<z.getIP()<<"i"<<endl;
    return (0);
}
```

In this program a concept LessThanComparable is defined on a type parameter T. This Concept requires that the type T defines the operation LessThan. The class Complex is a user defined type to represent the complex number. It contains attributes that represent the real part and the imaginary part for the complex number. It contains the usual set and get operations for each attribute. It defines the function abs() which returns the square root of the sum of the squares of the real part and the imaginary part.

The given concept map represents the way to tell the compiler that the users defined type Complex models the concept LessThanComparable. It gives an implementation for the required LessThan operation for a two complex parameters.

The class ArithmeticOperations is a template class with a generic type T constrained by a condition that the type T must model the LessThanComparable concept because the function member min calls the LessThan function on the two parameters a and b. In the main program the ArithmeticOperations class is instantiated by declaring an instance ArOp which takes Complex as its actual parameter. If complex did not model the concept LessThanComparable by the previously given concept map the compiler would have given an error on the declaration statement.

***The Proposed Algebraic Semantics of CDL:***

In this section, we give the algebraic semantics of CDL. We give the algebraic specification and interpretation rules for each construct of the language CDL. The specification rules are used to transform each construct into an algebraic specification. The interpretation rules are used to construct algebras to interpret the

specifications. First, we start by giving a group of definitions. Second, we give the rules to transform the concept construct to an algebraic specification. Third, we give the algebraic specification and interpretation rules for concept maps. Finally, we give the algebraic specification and interpretation rules of a generic class that uses concepts to constraint its parameter. We give a proof for each property the semantic model satisfies. We define the specification of the concepts, concept maps and template classes using the methodology of transformation rules defined by Fronk Fronk, A. (2002).

### *4.1: General Definitions:*

***Notations:***

A specification S of a construct C is denoted by the symbol spec(C).
The sorts of S is denoted by the symbol sorts(S) or sorts(spec(C)).
The operations of S is denoted by the symbol opns(S) or opns(spec(C)).
The axioms of S is denoted by the symbol axms(S) or axms(spec(C)).
Let S be a parameterized specification

### *4.2: Transformation Rules for Concepts:*

Each CDL concept is transformed into an algebraic specification. We state the following rules for concept transformation:

***Transformation Rule 1:***

1. The transformation of the parameter must be done before the transformation of the concept.
2. A concept cp with a type parameter ctp that is constrained by a concept cp is transformed into a parameterized specification CP carrying the name of the concept in capital letters, where CP= λCTP.CSP, CSP is the specification that is obtained by the transformation of the operations of cp.
3. The transformation of the parameter CTP is imported into CSP.

***Transformation Rule 2:***

A method signature in the concept cp of the form mt mv(t1 p1,….,tn pn) is transformed into an operation in opns(CSP) of the form mv : $t1 \times \ldots \times tn \rightarrow mt$ .

***Example 1:***

The following concept:

```
concept LessThanComparable <typename T>
{
bool LessThan(T a,T b);
}
```

is transformed into the specification:

| LESSTHANCOMPARABLE = λT.CSP |
|---|
| where<br>CSP= **import** T<br>**opns**<br>LessThan: T x T → bool |

### *4.3 Transformation Rules for Concept Maps:*

A concept map must actualize the concept definition and give an implementation for each method in the concept definition. So, to transform a concept map, we have to give the definition of the concept actualization first.

***Definition 1:***

Let cp be concept, CP = λCTP.CSP is its transformation. An actualization of a concept CP with an actual type ACTP is defined as:

CP(ACTP) = SP[CTP/ACTP]

which is the replacement of the formal parameter CTP with the actual parameter ACTP.
We now give the transformation rules of concept maps.

***Transformation Rule 3:***

1. The transformation of the actual parameter must be done before the transformation of the concept map.
2. A concept map for a concept cp with a formal parameter ctp on a actual type actp is transformed into a flat specification with the name CM(CP[CTP/ACTP]).
3. The transformation of the parameter ACTP is imported into CSP.

***Transformation Rule 4:***

1. A method signature in the concept map cp<actp> of the form mt mv(t1 p1,….,tn pn) is transformed into an operation in opns(CM(CP[CTP/ACTP])) of the form mv : $t1 \times \ldots \times tn \rightarrow mt$ .
2. Each methods implementation is transformed into adequate axioms.

***Example 2:***

The following concept map:

```
concept_map LessThanComparable<Complex>
{
  bool LessThan(Complex a, Complex b)
  {
   return(abs(a)<abs(b));
  }
}
```

is transformed into the specification as follows:

CM (LESSTHANCOMPARABLE[T/COMPLEX]) =
**sorts**
bool, complex
**opns**
LessThan: Complex × Complex → bool
**vars**
a,b: Complex
**axms**
LessThan(a,b) = (abs(a) < abs(b))

***Interpretation Rules of Concept Maps:***

Let cp be a concept, CP= λCTP.CSP is its transformation, and let cp<actp> be a concept map that models the concept cp for the type actp, CM(CP[CTP/ACTP]) its transformation. Let AACTP be a ∑- ACTP algebra which models ACTP. Let ACP_ACTP be a ∑-CM(CP[CTP/ACTP]) algebra, CM(CP[CTP/ACTP]) is interpreted by ACP_ACTP as follows:

***Interpretation Rule 1:***

ACP_ACTP interprets each operation in CM(CP[CTP/ACTP]) of the form a: s1×…×sn → s where s1,…,sn, s are members of sorts(CM(CP[CTP/ACTP])) arbitrarily.

***Interpretation Rule 2:***

ACP_ACTP Interprets each operation in opns (ACTP) is as in AACTP. that is :

$\forall f: s1 \times \ldots \times sn \rightarrow s \in opns(\text{ACTP}), \forall a \in \text{ACTP}s1 \times \ldots \times \text{ACTP}sn \rightarrow \text{ACTP}s$:
$f^{\text{ACP\_ACTP}}(a) = f^{\text{AACTP}}(a)$

The following lemma proves that the semantic algebra that interprets the concept map preserves the interpretation of the class when it is used as a sort for the parameters in the concept map.

***Lemma 1:***

Let cp be concept, ctp its parameter, CP= λCTP.CSP its transformation, let actp be a class, ACTP its transformations, let cp<actp> be the concept map which implements CP for type ACTP, CM(CP[CTP/ACTP]) is its transformation. CP, ACTP are achieved by applying the transformation rules. Let ACP_AACTP, AACTP be the ∑-CSP[CTP/ACTP] algebra, ∑-ACTP algebra respectively that results from the interpretation rules, then ACP_AACTP $|_{\sum\text{-ACTP}}$ is a model of ACTP.

***Proof:*** sorts(ACTP) contains a sort actp. By transformation rule 1 (part 3) Since ACTP is imported into CSP then sorts(CSP) also contain the sort actp and sorts(ACTP) sorts(CSP), opns(ACTP) opns(CSP). By interpretation rule 1 ACP_AACTP contains an interpretation for act. By interpretation rule 2 each operation in opns(ACTP) is interpreted in ACP_AACTP . Since ACP_AACTP is a model for ∑- CSP [CTP/ ACTP] and contains the interpretation for all of the sorts and operations of ACTP as interpreted in AACTP, which means that ACP_AACTP $|_{\sum\text{-ACTP}\_}$ is a model of ACTP.■

***Example 3:***

In this example, the concept map LessThanComparable<Complex> is instantiated using the parameter Complex, we give the interpretation of the specification LESSTHANCOMPARABLE_COMPLEX.. The specification CM(LESSTHANCOMPARABLE[T/COMPLEX]) is interpreted by the ∑-CM (LESSTHANCOMPARABLE[T/COMPLEX]) algebra LESSTHANCOMPARABLE_COMPLEX, The method LessThan is interpreted in LESSTHANCOMPARABLE_COMPLEX as follows:

$$LessThan^{\text{LESSTHANCOMPARABLE\_COMPLEX}}(a,b) = (abs^{\text{LESSTHANCOMPARABLE\_COMPLEXCOMPLEX}}(a) < abs^{\text{LESSTHANCOMPARABLE\_COMPLEXCOMPLEX}}(b))$$

Where the operation abs is interpreted in LESSTHANCOMPARABLE_COMPLEX as in COMPLEX

$$abs^{\text{LESSTHANCOMPARABLE\_COMPLEX}}(a) = abs^{\text{COMPLEX}}(a)$$

$$abs^{\text{LESSTHANCOMPARABLE\_COMPLEX}}(b) = abs^{\text{COMPLEX}}(b)$$

***4.4 Transformation Rules for a Constraint Template Class:***

The transformation of a constrained template class is related to two cases of instantiation. The first case is when the actual parameter satisfies the constraint associated with the template class. In this case the instantiation can be done successfully. The transformation in this case is similar to the unconstraint template class taking into consideration that the implementation of the constraining concept must be included in the specification. The second case occurs when an instantiation cannot be done because the actual parameter does not satisfy the class constraint. In this case, an error specification that describes the instantiation error is instantiated instead. We do not handle error specification at this stage. We leave it for a further work.

Assuming that the template class is going to be provided at instantiation time by an actual parameter that satisfies the constraining concept associated with the template class, the constrained template class is transformed into a parameterized specification using the following rules:

***Transformation Rule 5:***

1. The transformation of the parameter and the constraining concept must be done before the transformation of a constrained template class.
2. A template class c with a type parameter tp that is constrained by a concept cp is transformed into a parameterized specification C carrying the name of the class capital letters, where C = λTP.SP, SP is the specification that is obtained by the transformation of the attributes and the operations of c.
3. The transformation of the parameter TP is imported into SP.
4. The transformation of the actualization of the concept transformation through the type tp (i.e. CP[TP/CTP] is imported into SP.
5. The sort tp is added to sorts(SP).

Naturally, there is no information about the parameter at the time of template class definition, so, the transformation of the parameter actually consists of set of sorts with a single element with name p. Also, both the set of operations and axioms are empty. The transformation of the constraining concept contains an empty set of axioms since there is no implementations of the operations.

***Transformation Rule 6:***

1. A method signature in the class c of the form mt mv(t1 p1,….,tn pn) is transformed into an operation in opns(SP) of the form mv : $c \times t1 \times \ldots \times tn \rightarrow mt$ .
2. Each identifier mt, t1,.., tn is added to sorts(SP).

***Transformation Rule 7:***

1. An attribute declaration of the form atp a; where atp is another class; is transformed into an operation of the form**:**

**a:** $c \rightarrow a$and added to opns(C).

2. The transformation of the class atp is imported into C.

The decision whether the class can be instantiated correctly or not is done at instantiation time, i.e it cannot be known until we provide the class with an actual parameter. So, the compiler must check whether the actual parameter models the constraining concept or not. Checking modeling in terms of algebraic specification is done by first checking that the specification of the concept map that describes the modeling is defined and secondly by checking that this concept map specification actually contains axioms for all the operations of the corresponding concept on the provided actual data type. So, to perform the transformation of the actualization we need to define formally what does it mean to say that a data type d with the specification D models a concept cp with the specification CP.

***Definition 2:***

Let SP be a specification, the set opnam(SP) denotes the set of all operation names of SP, that is:

opnam(SP) = $\{f | f: s1 \times \ldots \times sn \rightarrow s \in opns(Sl.$

The set axmnam(SP) denotes the set of all axiom names of SP, that is:

axmnam(SP) = $\{f |\ f(a1, \ldots, an) = L[a1, \ldots, an] \in axms(SP)\ , f: s1 \times \ldots \times sn \rightarrow s \in opns(Sl.$

***Definition 3:***

A class d with transformation D is said to model a concept cp with transformation CP = λCTP.CSP if and only if the following two conditions are satisfied:

1. Condition1:

$$opns(CM(CP[CTP/D])) - opns(CP[CTP/D]) = \phi$$

2. Condition 2:

$$opnam(CP[CTP/D]) - axmnam(CM(CP[CTP/D])) = \phi$$

That is, a type is said to model a concept if and only if this type value implements all the operations of that concept. This means that the concept map of that type value must define all the operations of the corresponding concept and must give an implementation for each operation.

Now, the transformation of an actualization of a constrained template class can be given by the following transformation rules:

***Transformation Rule 8:***

If c is a template class constrained by the concept cp such that its transformation C = λTP.SP then:

1. The transformation of the actual parameter acp must be done before the transformation of the actualization.
2. The transformation of the actualization C[TP/ACP] is done by the replacement of each occurrence of the formal parameter TP by the actual parameter ACP. The sort tp is also replaced by acp.
3. An actualization of the template class of the form c<acp> is checked before transformation. If the type acp models the concept constraining the class c then c<acp> is transformed into C[TP/ACP] (an actualization of the specification C). If the type acp does not models the concept constraining the class c then c<acp> is transformed into an actualization of an error specification that describes the transformation error.

***Example 4:***

The following class:

```
template <typename tp> where LessThanComparable<tp>
class UseMin
{
  tp  min(tp u, tp v)
  {
     return(LessThan(u,v)?u:v);
  }
}
```

is transformed into the specification:

| USEMIN = λTP.CSP |
|---|

CSP = **import** TP, LESSTHANCOMPARABL[CTP/TP]
**sorts**
TP
**opns**
min: TP × TP → TP
**vars**
u,v: TP
**axms**
min(u,v)=(LessThan(u,v)?u:v)

***Interpretation Rules of the Actualization of the Constrained Parameterized Classes:***

Assume that c is a constrained parameterized class, C is its transformation, let AC be a ∑-C algebra, C is interpreted as follows:

***Interpretation Rule 3:***

1. AC interprets each sort in C arbitrarily.
2. AC interprets each attribute in C of the form a: c → s where s is a member of sorts(C) arbitrarily.
3. AC interprets each operation in C of the form a: c×s1×…×sn → s where s1,…,sn, s are members of sorts(C) arbitrarily.

***Interpretation Rule 4:***

If E is an element class that is used as a sort for parameters or attributes in C, let AE be a ∑-E then:

AC Interprets each operation in opns (E) is as in AE. that is :

$$\forall f: s1 \times \ldots \times sn \to s \in opns(E), \forall a \in Es1 \times \ldots \times Esn \to Es:$$

$$f^{AC}(a) = f^{AE}(a)$$

***Interpretation Rule 5:***

If P is the parameter class that is used as a sort for parameters or attributes in C, let AP be a ∑-P then: AC Interprets each operation in opns (P) is as in AP. that is:

$$\forall f: s1 \times \ldots \times sn \to s \in opns(P), \forall a \in Ps1 \times \ldots \times Psn \to Ps:$$
$$f^{AC}(a) = f^{AP}(a)$$

***Interpretation Rule 6:***

Let cp be the constraining concept of the parameterized class, let ACP_ACTP be a ∑-CM(CP[CTP/ACTP]) algebra then:

AC Interprets each operation in opns (CM(CP[CTP/ACTP]) ) is as in AACTP. that is:

$$\forall f: s1 \times \ldots \times sn \to s \in opns(\mathrm{CM(CP[CTP/ACTP]}),$$
$$\forall a \in \mathrm{ACP\_ACTP}s1 \times \ldots \times \mathrm{ACP\_ACTP}sn \to Ps:$$
$$f^{AC}(a) = f^{\mathrm{ACP\_ACTP}}(a)$$

Now, we present the following lemma to prove that our semantic model preserves the interpretation of the class when it is used as a parameter in a generic class.

***Lemma 2:***

Let c be a generic class p its parameter, cp its constraining concept on parameter p, PSC = λP.SC its transformation, let act be a class such that act models cp, ACT its transformations, PSC, ACT are achieved by applying the transformation rules. Let AC, AACT be the ∑-PSC[P/ACT] algebra, ∑-ACT algebra respectively that results from the interpretation rules, then $AC|_{\sum\text{-}ACT}$ is a model of ACT.

***Proof:*** PSC, ACT are achieved by following the presented transformation rules and due to transformation rule 5 (part 5) sorts(P) contains a sort p. By transformation rule 5 (part 3) Since P is imported into SC then sorts(SC) also contain the sort p and sorts(P) sorts(PSC), opns(P) opns(PSC). From the definition of actualization PSC[P/ACT] replaces each occurrence of P by ACT. By transformation rule 8 part 3, PSC[P/ACT] replaces p by act. . By interpretation rule 3 AC contains an interpretation for act. By interpretation rule 5 each operation in opns(ACT) is interpreted in AC. Since AC is a model for ∑-PSC[P/ACT] and contains the

interpretation for all of the sorts and operations of ACT as interpreted in AACT , which means that $AC|_{\sum\text{-}ACT}$ is a model of ACT. ■

The following lemma shows that the model preserves the interpretation of the concept map when it is used as a constraint in a generic class instance of an actual parameter.

***Lemma3:***

Let c be a generic class p its parameter, cp its constraining concept on parameter p, let PSC = λP.SC be the transformation of the class, let actp be a class such that act models cp, ACTP its transformations, let cp<actp> be the concept map of cp over ACTP, CM(CP[CTP/ACTP]) its transformation of the concept map, PSC, CM(CP[CTP/ACTP]) are achieved by applying the transformation rules. Let AC, ACP_ACTP be the ∑-PSC[P/ACTP] algebra, ∑- CM(CP[CTP/ACTP]) algebra respectively that results from the interpretation rules, then $AC|_{\sum}$- CM(CP[CTP/ACTP]) is a model of CM(CP[CTP/ACTP]).

***Proof:*** By transformation rule 5 (part 5) sorts(P) contains a sort p. By transformation rule 5 (part 3) Since P is imported into SC then sorts(SC) also contain the sort p and sorts(P) ⊆ sorts(PSC), opns(P) ⊆ opns(PSC). From the definition of actualization PSC[P/ACTP] replaces each occurrence of P by ACT. By transformation rule 8 part 3, PSC[P/ACTP] replaces p by act, so when the generic class is actualized, the specification P is replaced by ACTP, and since CP[P/CTP] the transformation of the actualization of the concept transformation through the type p is imported into SP, by actualization P is replaced by ACTP. By interpretation rule 3 AC contains an interpretation for act. By interpretation rule 6 each operation in opns(CM(CP[CTP/ACTP])) is interpreted in AC . Since AC is a model for ∑-PSC[P/ACTP] and contains the interpretation for all of the sorts and operations of CM(CP[CTP/ACTP] as interpreted in ACP_ACTP, which means that $AC|_{\sum}$- CM(CP[CTP/ACTP]) is a model of CM(CP[CTP/ACTP]. ■

***Example 5:***

Assume that the class UseMin is instantiated over the parameter complex (i.e UseMin<Complex> ). Here we give the interpretation of the specification USEMIN. We must first give the interpretation of the concept map for the constraining concept LessThanComparable on the parameter Complex.

The specification CM(LESSTHANCOMPARABLE[T/COMPLEX]) is interpreted by the∑-CM(LESSTHANCOMPARABLE[T/COMPLEX]) algebra LESSTHANCOMPARABLE_COMPLEX, usemin is interpreted arbitrarily by interpretation rule 3. The method LessThan is interpreted in USEMIN as follows:

$$LessThan^{\text{LESSTHANCOMPARABLE\_COMPLEX}}(a,b) = (abs^{\text{COMPLEX}}(a) < abs^{\text{COMPLEX}}(b))$$

The specification USEMIN is interpreted by the ∑- USEMIN algebra USEMIN, usemin is interpreted arbitrarily by interpretation rule 3. The method min is interpreted in USEMIN as follows:

$$min^{\text{USEMIN}}(um,a,b) = if\,(LessThan^{\text{USEMIN}}(a,b)),a,b)$$
$$LessThan^{\text{USEMIN}}(a,b) = LessThan^{\text{LESSTHANCOMPARABLE\_COMPLEX}}(a,b)$$

***Conclusion:***

In this paper, we defined a mini-language we called CDL which is of object oriented nature that contains the basic constructs of constrained generic programming. It contains the simple concept, concept map and constrained template class. All of these constructs takes only one type parameter in order to simplify the problem and the semantics description of these constructs. The concept can define a group of function signatures as concept requirements. The basic contribution here is that we have presented transformation rules to transform these constructs into algebraic specification. Also, we presented interpretation rules to transform these algebraic specifications into algebras. A CDL concept is transformed into a parameterized specification. The function signatures are transformed into operations.

For concept maps, we gave rules for transforming concept maps on a specific type into a simple specification in which the function headers are transformed into operations and the function bodies are described using axioms. Since the concept map contains functions with implementation bodies, they need to be described by transforming the algebraic specification into an algebra that models it. We gave interpretation rules for building this algebra that adopt the loose-semantics approach in order to give the implementer the freedom to choose the suitable method for interpreting the imperative parts.

One of the important issues of studying concepts is to be able to check whether an actual type models the concept or not. This is a basic issue in checking template classes constrained by a concept. We gave a formal definition for type-concept-modeling using the transformed specifications of both concept and concept maps. For the constrained generic class, we gave the transformation rules to transform it into a parameterized

specification. The member variables and member functions are both transformed into operations. Assuming that the type models the concept, all its implemented operations are interpreted in the constrained class.

We proved by lemmas (1, 2 and 3) that the concept map specification preserves the semantics of the parameter. We also proved that when the parameterized class is transformed after being type-checked using concept modeling type checking that it preserves the semantics of its parameter operations and the semantics of the operation of the constraining concept.

## REFERENCES

Aho. A.V., R. Sethi and J.D. Ullman, 1996. *Compilers: Principles, Techniques and Tools*, Addison-Wesley.

Austern, M.H., 1998. *Generic programming and the STL: Using and extending the C++ Standard Template Library. Professional Computing Series*. Addison-Wesley.

Broy, M. and M. Wirsing, 1982. Algebraic definition of a functional programming language. *IEEE Transactions on Information Theory*, 17(2): 137-161..

Broy. M., M. Wirsing and P. Pepper, 1987. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1): 54-99.

Florent, K. and S. Fransois-Regis, 2007. Rule-Based Operational Semantics for an Imperative Language. *Electronic Notes In Computer Science* 174: 35-47.

Fronk, A., 2002. *An Approach To Algebraic Semantics Of Object-Oriented Languages*. Memorandum 129, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universit ¨at Dortmund

Gregor, D., J. Jarvi, J. Siek, B. Stroustrup, G.D. Reis and A. Lumsdaine, 2006. Concepts: Linguistic Support for Generic Programming in C++. In *21st Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06),* October 22-26, Portland, Oregon, USA.

Harmen, K., K. Anneke and R. Arend, 2006. Defining Object Oriented Execution Semantics Using Graph Transformations. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006), LNCS.,* 4037: 186-201,

Jazayeri, M., R. Loos, D. Musser and A. Stepanov, 1998. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, April.

Mosses, P.D., 2006. Formal Semantics of Programming Languages - An Overview . *Electronic Notes In Theoretical Computer Science*, 148.

Pratt, T.W. and M.V. Zelkowitz, 2001. *Programming languages: Design and Implementation.* Prentice-Hall.

Slonneger, K. and B.L. Kurtz, 1995. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based. Approach*. Addison-Wesley.

Stroustrup, B., 1994. *Design and Evolution of C++*. Addison-Wesley.

Stroustrup, B., 2000. *The C++ Programming Language*. Addison-Wesley.

Tang, X. and J. Jarvi, 2007. Concept-based optimization. In *ACM SIGPLAN Symposium on Library-Centric Software Design. (LCSD'07)*, October 21-25, Montreal, Canada.

Wilhelm, R. and D. Maurer, 1995. *Compiler Design Reading*, Addison-Wesley.