An Algebraic Semantics Model for Templates Constrained with a Single Associated Type Concept

1Bashar Abu Sa and 2Abdullah Mohd Zin

1Center for Software Technology and Management, Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia.
2Center for Software Technology and Management, Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia.

ABSTRACT

Constraining templates with concepts is an efficient approach or providing early type checking of generic parameters in C++. The formal semantics of the concept constrained templates is important for a more accurate definition and understanding. In this paper we present an algebraic semantics model for concepts with associated types which constrains the template class. The model defines systematic transformation and interpretation rules of the constructs. The rules are used to transform the concept, concept map and constrained into parameterized specifications and interpret the specifications using algebras.

INTRODUCTION

Generic programming is a programming technique for developing software libraries that has efficiency and reusability (Jazayeri, 1998). C++ has its way to support generic programming by using the template classes that enables programmers to define generic type parameters. The problem that arises when using template classes is that the type checking of them cannot be performed until they are instantiated because they contain a generic type parameter that the compiler has no information about. As a solution to this problem, concepts where suggested by (Gregor, 2006) to be used for generic programming in C++. They are used to give constrains of the generic type parameters of a template class. Therefore using concepts type checking of a template class can be done at compilation time.

Our work begun by presenting an algebraic semantics model for concepts of a simple language called CDL. This language defines simple concepts that take one parameter and a set of function signatures. The description of CDL is given in (Abu Sa’, 2012). In this paper we attempt to give an algebraic semantics of an extension of CDL that defines the concept with the additional feature of associated type. We define a mini language CDLAT that contain the constructs of concept, concept map and template classes constrained by concepts. The concepts of CDLAT can define one type parameter, one associated type and a group of operations. Concept map can have one actual type parameter and one type definition and a group of implemented operations. The template class can have one type parameter. Our approach gives the algebraic specification of each construct by defining the transformation rules. We use the algebraic specification with explicit and implicit parameters for concepts, simple specification for concept maps and explicit parameterized parameters for template classes constrained by concepts. Interpretation rules to transform the specification of the concept maps and template class instantiation into algebras are also given. A proof of some related properties of concept that this model satisfy are presented and proved.

We use the structured algebraic specification to reflect the structures that are described. We elaborate the transformation rules mechanism to uniformly convert concepts into a formal specification which uses the parameterized specification. This is a feature that distinguishes our approach from others we use algebraic specification to describe concepts and associated type which is very efficient in formal description of constrained generic programming constructs and revealing their properties. Also, algebraic specification is adequate in describing concepts and concept maps because of their definition interface-like constructs that describes type requirements that are used as constraints in generic programming.

Corresponding Author: Bashar Abu Sa, Center for Software Technology and Management, Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia.
E-mail: BasharAS.Research@gmail.com

Keywords: Concepts, Associated Types, Algebraic Semantics, Generic Programming, Template Classes.
This paper is divided into 7 sections. In section 2 we give the related work. In section 3 we give a brief introduction on the fundamentals of algebraic semantics. In section 4 we define the CDLAT language. In section 5 we give the algebraic specification and semantics of CDLAT. In section 6 we present the proof of the correctness of the semantic model. In section 7 we give a conclusion and further work.

**Related Work:**

The formal semantics (Fronk, 2002), (Jazayeri,1998),(Harman, 2006), (Mosses, 2006), (Broy, 1982), (Broy, 1987), (Slonneger, 1995) has its important benefit for compiler (Aho, 1996), (Wilhelm, 1995) and programming languages design and implementation (Pratt, 2001). Algebraic semantics is an approach of formal semantics that were used to describe the formal semantics of functional, imperative and object oriented languages (Florent,2007), (Broy, 1982), (Stroustrup, 1994), (Harmen2006).

In C++ (Broy, 1987), (Stroustrup2000), (Jarvi, 2009), (Pirkelbauer,2009), (Reis, 2006) generic programming (Jazayeri,1998) is supported by template classes(Austern,1998). In (Gregor, 2006), the concept construct was presented as a new language feature for C++. The main purpose was to give a method for modular type checking for generic classes of C++. The main purpose was to give a method for modular type checking for generic classes of C++ (template classes) which represents the base for generic programming in C++. In (Reis, 2006)an informal specification of the concepts was given to be used by programmers and to enable checking of templates.

This feature also provided a lot of benefits to programming. It was used in type checking and compiler optimization as in (Tang, 2007). It was used to make library-defined optimization using mathematical properties as in (Gottschling, 2008). Also, it was used in designing a functional style programming of C++ as in (Jarvi, 2008).In (Gottschling, 2008), the concepts were used to constraint the generic functions of C++ based on mathematical properties in order to be used to enable library defined optimization that only requires a concept based compiler. In(Jarvi, 2008), a description of a design and implementation of a built in lambda functions in C++ using concept in order to provide C++ with functional style programming. It describes the way to infer the types of the parameters using the generic functions constrained by concepts which provides a modular type checking for the defined lambda functions. In (Bagge, 2009) a method was shown for using concept for automated unit testing. In our work we concentrate on using the algebraic semantics to give a formal description of the concept construct with the feature of defining an associated type. We also give the semantics of the concept map that models this concept on a concrete data type in addition to the template class constrained by that concept.

**The Language CDLAT:**

In this section we define a sublanguage CDLAT which adopts the C++ syntax to define the concept, concept map and template classes with one type parameter. More details about the syntax and semantics of concept can be found in(Gregor 2006).

**Concepts:**

A concept is a construct that describes what requirements that a type parameter should satisfy to be used in certain generic algorithm. A concept of CDLAT can have on type parameter and one associated type and a group of operations to be implemented by a type that models that concept. A concept in CDLAT can be defined as:

```
concept ConName<typename TP> { 
  typename IP;    //group of operations of the form
  rv mf(t1 p1, ..., tpn);
}
```

**Concept Maps:**

A concept map is a construct that is used to give the compiler the information that the given type models the corresponding concept and to describe how it models it. A CDLAT concept map is defined as follows:

```
concept_mapConceptName<TypeName> 
{  
typedef AIP IP;
    //A group of operations of the form
  rv mf(t1 p1, ..., tpn)
} 
```
{ // an implementation body
}

The typedef section in a concept map is considered as an implementation of the corresponding associated type in the concept. The operations are implementation of the corresponding operations of the concept definition.

**Template Classes Constrained by Concepts:**

A generic class can have one generic type parameter. It can use concepts to specify some conditions on the properties of the actual type parameters that can replace that formal parameter. A CDLAT generic class is defined as follows:

template <typename TP> where ConceptName<TP>

class ClassName
{
    //Group of attributes
    //Group of operations
}

**An Example Program:**

To explain the constructs of the CDLAT, we present the following simple program example:

```cpp
const intOutOfRange = -1;
const float PI = 3.14;

concept ValueComparable<typename TP>
{
    typename TI;
    bool CompareValue(TP a,TI b);
}

class Rectangle {
    float Length;
    float Width;
    public:
    Rectangle(float l, float w){Length=l; Width=w;}
    Rectangle(){Length=1; Width=1;}
    void setLength(float l){Length=l;}
    void setWidth(float w){Width=w;}
    float getLength() {return (Length);}
    float getWidth(){return(Width);}
    float Area(){ return(Length*Width);}
};

class Circle {
    float Radius;
    public:
    Circle(float r){ Radius = r;}
    Circle(){ Radius = 1;}
    void setRadius(float r){ Radius=r;}
};
```
float getRadius() {return (Radius);}
float Area() {return((Radius * Radius * PI));}

concept_mapValueComparable<Circle>
{
typedf Rectangle TI;

bool CompareValue(Circle a, TI b)
{
    return(a.Area() > b.Area());
}
}

template<typename T> where ValueComparable<T>
class GeometricProcesses
{
    float GetAreaDifference(T u, Rectangle v)
    {
        If (Compare(u, v)) { return(u.Area() - v.Area());}
        else { return(OutOfRange);}
    }
}

int main()
{
    Circle C(7);
    Rectangle R(5, 3);
    GeometricProcesses<Circle> GP;
    If (GP.GetAreaDifference(C, R) >= 0)
        cout << "The Area Difference :
        " << GP.GetAreaDifference(C, R) << endl;
    else cout << "The Area Exceeds the Limits" << endl;
    return 0;
}

This program defines a single parameter concept ValueComparable. This concept defines an associated type TA which must be actualized by any concept map that models this concept. It also defines an operation Compare that has two arguments of type TP and TA respectively. This operation must be implemented by any concept that models the concepts.

The class Rectangle defines two data members Length and Width of type float, a two-argument constructor, a default constructor and a set and get operations for each member variable. The function Area() returns the value of the area of the rectangle. The class Circle has one member attribute Radius of type float. It defines a single parameter constructor, a default constructor and set and get operation for the data member. The function Area() returns the area of the circle.

The concept map is used to tell the compiler that the type circle models the value comparable concept and it describes the way it models it. The concept map defines a type definition that models the associated type in the corresponding concept. The type TA is defined as a Rectangle. The concept map implements the operation compare value which takes two parameters a and b for type circle and TA respectively. This operation is implemented to return the result of the comparison between the two areas of a and b. The GeometricProcesses class is a template class with a single parameter T that is constrained by the concept ValueComparable. That is, this class cannot be instantiated by any type class unless it satisfies the ValueComparable concept. This class defines an operation GetAreaDifference() that takes two input arguments a and b of types Circle and Rectangle respectively. It uses the compare value function to compare the two area values of a and b and returns the positive difference between them if the area of a is larger than b. It returns an Out of Range value otherwise.

The main program defines an object C of type Circle with radius 7. It defines an object R of type Rectangle of Length 5 and width 3. It defines an instance GP of GeometricProcesses. It calls the member function GetAreaDifference of GP on the two objects C and R to print the values of area difference.
The Fundamentals of Algebraic Semantics:

In this section, we introduce some of the basics of algebraic specification which were introduced in (Fronk, 2002), (Slonneger, 1995). The model classes that we define in this paper can be considered as sets.

A signature is a tuple \( \Sigma := (S, \Gamma) \), where \( S \) is a set of sorts, \( S = \text{sorts}(\Sigma) \), and \( \Gamma \) is a set of operation symbols, \( \Gamma = \text{opns}(\Sigma) \). Each operation symbol is assigned a characteristic over \( S^* \times S \). Variables are \( S \)-sorted. \( X = \{X_s\} s \in S \) denotes a \( S \)-indexed family of sets \( X_s \) of variables for each \( s \in S \). The set of \( S \)-sorted \( \Sigma \)-terms over \( s, \text{Term}(\Sigma, X) \) for short, is defined as usual. An algebraic specification, \( \text{SP} \), is a tuple \( \langle \Sigma, E \rangle \) consisting of a signature, \( \Sigma = \text{sig}(\text{SP}) \), and a set of formulas over \( \Sigma, E \), \( \text{axms}(\text{SP}) \), called axioms. A specification \( \langle \Sigma', E' \rangle \) is called a sub specification of \( \langle \Sigma, E \rangle \), \( \Sigma' \subseteq \Sigma \), \( E' \subseteq E \), for short, if \( \Sigma' \subseteq \Sigma \) and \( E' \subseteq E \) hold.

A \( \Sigma \)-algebra, \( A \), is a pair \( \langle \{A_s\} s \in S, \{f^A\} f \in \Gamma \rangle \) consisting of a family \( \{A_s\} s \in S \) of non-empty carrier-sets. \( A \), for each \( s \in S \), and a set \( \{f^A\} f \in \Gamma \) of operations \( f^A : A_s \times \ldots \times A_s \rightarrow A \) for each \( f : s \times \ldots \times s \rightarrow s \in \Gamma \). \( A \) is a model of a specification, \( \langle \Sigma, E \rangle \), if it satisfies each formula \( e \in E \). The set of all models of \( \langle \Sigma, E \rangle \) is denoted by \( \text{Alg}(\Sigma, E) \). The loose semantics of a specification \( \text{SP} = \langle \Sigma, E \rangle \), \( \text{Mod} \text{(SP)} \) for short, is defined as the set \( \text{Alg}(\Sigma, E) \).

Let \( \Sigma = \langle S, \Gamma \rangle \), and \( \Sigma' = \langle S', \Gamma' \rangle \) be two signatures with \( \Sigma' \subseteq \Sigma \). and let \( A' = \text{sig}(\Sigma') \). The \( \Sigma \)-algebra \( A' \models A \) is called a \( \Sigma' \)-reduct of \( A \), if for each \( s \in S \) the carrier-set \( (A' \models A)_s \) is defined as \( A_s \)'s, and for each \( f \in \Gamma \) the operation \( f^A \subseteq \Sigma \) is defined as \( f^A \). Let \( SP \) and \( SP' \) be two specifications, such that the set pairwise equal operation symbols have the same characteristics. The specification-building operation import into: \( \text{SPEC} \times \text{SPEC} \rightarrow \text{SPEC} \) is defined by \( \text{sig(import SP into SP')} := \text{sig}(\text{SP}) \cup \text{sig}(\text{SP'}) \) and by \( \text{Mod(import SP into SP')} := \{A \in \text{Alg}(\text{import SP into SP'}) | \text{A} \models \text{Sig}(\text{SP}) \} \). Let \( SPA = \langle \Sigma, E \rangle \) be a specification, and let \( SPB \) be a sub specification of \( SPA \). The pair \( HS = \langle \Sigma, E \rangle \) is called hierarchical specification. \( SPA \) is called subclass specification, \( SPB \) is called superclass specification of \( HS \). Let \( \text{sig} \text{(HS)} = \Sigma \). The loose semantics of \( HS \) is defined as: \( \text{Mod} \text{(HS)} := \{A \in \text{Mod}(SPA) | A \models \text{Sig}(SPB) \} \).

The Algebraic Semantics of CDLAT:

In this section, we give the algebraic semantics of CDLAT. We give the algebraic specification and interpretation rules for each construct of the language CDLAT. 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. The basic constructs in the language are the concepts with associated types, the concept maps with type definitions and the template classes constrained by a concept with an associated type. For the concept, we give the rules to transform it construct to a parameterized algebraic specification. For the concept map, we give the transformation rules to transform it into algebraic specification and we give interpretation rules to construct an algebra that interprets the specification. We give the algebraic specification and interpretation rules of a generic class that uses concepts to constraint its parameter. We also give a proof for some properties the semantic model satisfies. The meaning of the constrained generic programming constructs with concepts of C++ can be found in (Gregor, 2006). The methodology of the transformation and interpretation rules in algebraic semantics were used to describe object oriented constructs as in (Fronk, 2002).

General Definitions:

Notations:
A specification \( S \) of a construct \( C \) is denoted by the symbol \( \text{spec}(C) \).
The sorts of \( S \) is denoted by the symbol \( \text{sorts}(S) \) or \( \text{sorts} \text{(spec}(C)) \).
The operations of \( S \) is denoted by the symbol \( \text{opns}(S) \) or \( \text{opns} \text{(spec}(C)) \).
The axioms of \( S \) is denoted by the symbol \( \text{axms}(S) \) or \( \text{axms} \text{(spec}(C)) \).

If \( S \) is a parameterized specification of the form \( \text{PEISP} = \lambda \text{EPAR}. \{\text{IPAR}\}. \text{CSP} \) then the set of the explicit parameters is denoted by \( \text{ExpPar} \text{(PEISP)} = \{\text{EPAR}\} \). The set of the implicit parameters is denoted by \( \text{ImpPar} \text{(PEISP)} = \{\text{IPAR}\} \).

Transformation of concepts:
Each CDLAT concept that contains a parameter and an associated type is transformed into a parameterized algebraic specification (that contains implicit and explicit parameters). We state the following rules for concept transformation:

**Transformation Rule 1:**
1. The transformation of the explicit parameter must be done before the transformation of the concept.
2. The transformation of the associated type must be done before the transformation of the concept.
3. A concept \( cp \) with a type parameter \( ctp \) is transformed into a parameterized specification \( CP \) carrying the name of the concept in capital letters, where \( CP = \lambda \text{ctp}. \{\} \text{CSP} \). CSP is the specification that is obtained by the transformation of the operations of \( cp \). CTP represents the specification that results from the transformation of the parameter of the concept. The empty brackets represent an empty set of implicit parameters.
4. The transformation of the parameter CTP is imported into CSP.

**Transformation Rule 2:**
An associated type of the form typename cip; is transformed into an implicit parameter CIP and added to the set of the implicit parameters. CIP is imported to CSP.

**Transformation Rule 3:**
A method signature in the concept cp of the form mt mv(t1 p1,...,tnpn) is transformed into an operation in opns(CSP) of the form mv : t1 x ... x tn → mt .

**Example 1:**
The following concept:

```
concept ValueComparable<typename TP>
{
typename TI ;
boolCompareValue(TP a, TI b);
}
```

is transformed into the specification:

```
VALUECOMPARABLE = λTP.{TI}.CSP
where
CSP=import TP, TI
opns
CompareValue: TP x TI → bool
```

**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.{CIP}.CSP is its transformation. An actualization of a concept CP with an actual explicit type ACTP and actual implicit type ACIP is defined as:

```
CP(ACTP){ACIP} = CSP[CTP/ACTP]{ CIP/ACIP }
```

which is the replacement of the formal parameter CTP with the actual parameter ACTP and the implicit formal parameter CIP with the actual parameter ACIP.

The transformation rules of concept maps for concept with associated type are as follows:

**Transformation Rule 4:**
1. The transformation of the actual parameter must be done before the transformation of the concept map.
2. The transformation of the actual type in the type definition must be done before the transformation of the concept map.
3. 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]).
4. The transformation of the parameter ACTP is imported into CSP.

**Transformation Rule 5:**
1. A type definition of the form typedef ACIP CIP; is transformed by removing the implicit parameter CIP from the implicit parameters list. Every occurrence of CIP is replaced by ACIP.
2. The transformation of the parameter ACIP is imported into CSP.

**Transformation Rule 6:**
1. A method signature in the concept map cp<actp> of the form mt mv(t1 p1,...,tnpn) is transformed into an operation in opns(CM(CP[CTP/ACTP])) of the form mv : t1 × ... × tn → mt .
2. Each methods implementation is transformed into adequate axioms.
Here, in transformation rule 6 part 2, we only stated that: Each methods implementation is transformed into adequate axioms. In section 5.7 we will describe in detail how to integrate the imperative parts represented by the bodies of the operations and to transform them into axioms.

Example 2:
The following concept map:

```cpp
concept_mapValueComparable<Circle> {
    typedef Rectangle TI;
    bool CompareValue(Circle a, TI b) {
        return(a.Area() > b.Area());
    }
}
```

is transformed into the specification:

```
CM (VALUECOMPARABLE[TP/CIRCLE]) = import CIRCLE, RECTANGLE
sorts bool, circle, rectangle
opns CompareValue: Circle × Rectangle → bool
vars a:Circle, b:Rectangle
axms CompareValue(a, b) = (Area(a) > Area(b)) •
```

For the axiom, \(\text{CompareValue}(a, b) = (\text{Area}(a) > \text{Area}(b))\), the operations \(\text{Area}(a)\), \(\text{Area}(b)\) comes from the imported specification of circle and rectangle. In section 5.7 we give the details of transforming the imperative parts in the bodies of the operations into axioms.

Interpretation Rules of Concept Maps:
Let \(cp\) be a concept, \(CP = \lambda \text{CTP}. \{CIP\}. \text{CSP}\) is its transformation, and let \(cp<actp>\) be a concept map that models the concept \(cp\) for the type \(actp\), \(\text{CM}(CP[CTP/ACTP])\) its transformation. Let \(\text{AACP}\) be a \(\Sigma\)-\(\text{ACTP}\) algebra which models \(\text{ACTP}\). Let \(\text{ACP}_\text{ACTP}\) be a \(\Sigma\)-\(\text{CM}(CP[CTP/ACTP])\) algebra, \(\text{CM}(CP[CTP/ACTP])\) is interpreted by \(\text{ACP}_\text{ACTP}\) as follows:

**Interpretation Rule 1:**
\(\text{ACP}_\text{ACTP}\) interprets each operation in \(\text{CM}(CP[CTP/ACTP])\) of the form \(f: s_1 \times \ldots \times s_n \rightarrow s\) where \(s_1, \ldots, s_n, s\) are members of sorts(\(\text{CM}(CP[CTP/ACTP])\)) arbitrarily.

**Interpretation Rule 2:**
\(\text{ACP}_\text{ACTP}\) interprets each operation in \(\text{opns}(\text{ACTP})\) as in \(\text{AACP}\), that is:
\[
\forall f: s_1 \times \ldots \times s_n \rightarrow s \in \text{opns}(\text{ACTP}), \forall a \in \text{ACTP}s_1 \times \ldots \times \text{ACTP}s_n:

f_{\text{ACP}_\text{ACTP}}(a) = f_{\text{AACP}}(a)
\]

**Interpretation Rule 3:**
Let typename \(\text{ACIP}\ CIP;\) be a type definition in \(\text{CM}(CP[CTP/ACTP])\). Let \(\text{AAACIP}\) be a \(\Sigma\)-\(\text{ACTP}\) algebra which models \(\text{ACTP}\). Let \(\text{ACP}_\text{ACTP}\) be a \(\Sigma\)-\(\text{CM}(CP[CTP/ACTP])\) algebra, \(\text{CM}(CP[CTP/ACTP])\) is as in \(\text{AAACIP}\). that is:
\[
\forall f: s_1 \times \ldots \times s_n \rightarrow s \in \text{opns}(\text{ACIP}), \forall a \in \text{ACIP}s_1 \times \ldots \times \text{ACIP}s_n:

f_{\text{ACP}_\text{ACTP}}(a) = f_{\text{AAACIP}}(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 an actual type that implements an associated type in the concept map.
Lemma 1:
Let \( cp \) be concept, \( ctp \) its parameter, \( CP = \lambda CTP.\{CIP\}.CSP \) its transformation, let \( actp \) be a class, \( ACTP \) its transformations, let \( cp<actp> \) be the concept map which implements \( CP \) for type \( ACTP \) and contains a type definition \( typedefacipcip \); that implements the associated type \( cip \) of the concept \( cp \), \( CM(\{CP\[CTP/ACTP\]) \) is the transformation of the concept map, \( ACIP \) is the transformation of \( acip \), \( CP, ACIP \) are achieved by applying the transformation rules. Let \( ACP_AACTP, AACIP \) be the \( \Sigma -CM(\{CP\[CTP/ACTP\]) \) algebra, \( \Sigma -ACIP \) algebra respectively that results from the interpretation rules, then \( ACP_AACTP |\Sigma -ACIP \) is a model of \( ACIP \).

Proof:
Sorts(\( ACIP \)) contains a sort \( acip \). Transformation rule 4 (part 4) imports \( ACIP \) into CSP so, sorts(CSP) also contain the sort \( actp \) and sorts(\( ACIP \)) \( \subseteq \) sorts(CSP), ops(\( ACIP \)) \( \subseteq \) ops(CSP). By interpretation rule 1 \( ACP_AACTP \) contains an interpretation for \( acip \). By interpretation rule 3 each operation in ops(\( ACIP \)) is interpreted in \( ACP_AACTP \). Since \( ACP_AACTP \) is a model for \( \Sigma -CM(\{CP\[CTP/ACTP\]) \) and contains the interpretation for all of the sorts and operations of \( ACIP \) as interpreted in \( AACIP \), which means that \( ACP_AACTP |\Sigma -ACIP \) is a model of \( ACIP \).

Example 3:
In this example, the concept map \( ValueComparable<Circle> \) is defined using the parameter \( Circle \) and an associated type implemented by the actual \( Rectangle \). To give the interpretation of the specification \( VALUECOMPARABLE\_CIRCLE \), we must first give the interpretation of the concept map for the operations Area in \( ValueComparable \) on the parameter \( Circle \) and on the parameter \( Rectangle \).

The specification \( CM(VALUECOMPARABLE\_T/CIRCLE)) \) is interpreted by the \( \Sigma -CM(VALUECOMPARABLE\_T/CIRCLE) \) algebra \( VALUECOMPARABLE\_CIRCLE \).

The method Area is interpreted by interpretation rule 2 and 3. The method Area is interpreted in \( VALUECOMPARABLE\_CIRCLE \) as follows:

\[
Area^{VALUECOMPARABLE\_CIRCLE} (a) = (Area^{CIRCLE} (a))
\]

\[
Area^{VALUECOMPARABLE\_CIRCLE} (b) = (Area^{RECTANGLE} (b))
\]

CompareValue is interpreted by interpretation rule 1. The method CompareValue is interpreted in \( VALUECOMPARABLE\_CIRCLE \) as follows:

\[
CompareValue^{VALUECOMPARABLE\_CIRCLE} (a, b) = (Area^{VALUECOMPARABLE\_CIRCLE} (a) > Area^{VALUECOMPARABLE\_CIRCLE} (b))
\]

Transformation Rules of a Constraint Template Class:
When its parameter satisfies the constraint, a constrained template class is instantiated in the same way as the unconstrained class. The operations of the constraining concept are included in the specification of the template class. When the parameter does not satisfy the constraining concept, the class cannot be instantiated, and instead a specification that represents an error is instantiated. We do not handle error specification at this stage. We leave it for a further work.

Basically, the constrained template class is transformed to a specification based on the satisfaction of the constrained attached after the where clause. Consider the class:

```cpp
template <typename TP> where CN<TP>
class C
{
    //Group of attributes
    //Group of operations
}
```

This class is transformed into a conditional specification of the form:

\[
C = \begin{cases}
    \lambda TP, SP, & \text{if } TP \text{ models } CN \\
    SP\text{Error}, & \text{otherwise}
\end{cases}
\]

Note the condition (if \( TP \text{ models } CN \)) which we define formally in definition 3. This means that when the condition is true, the class is transformed into a parameterized specification as declared. If the condition is false
then it is transformed into a specification that describes the instantiation error. The details of the error case are
left for a future research. TP represents the specification of the parameter. SP represents the specification
obtained from the transformation of the attributes and operations of the class.

The following transformation rules can be used to construct the specification that results from the transformation
of the template class. These transformation rules process the case when the actual parameter at instantiation time
satisfies the constraint of the template class, that is:

\[ C = \lambda TP.SP. \]

**Transformation Rule 7:**

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 \( C = \lambda TP.SP \), 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).

At the time the template class is defined the information about the parameter is not available. This means
that at this stage that the operations and axioms of the parameter specification are not known. The
transformation of the parameter P consists of set of sorts with a single element with name p. The transformation
of the constraining concept contains an empty set of axioms since there are no implementations of the
operations.

**Transformation Rule 8:**

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

**Transformation Rule 9:**

1. An attribute declaration of the form atp a; where atp is another class; is transformed into an operation
   o of the form:
   \[ a : c \rightarrow atp \] and added to opns(C).
2. The transformation of the class atp is imported into C.

Before an object of parameterized class constrained by a concept can be instantiated over an actual
parameter, the actual parameter must be checked first whether it models the constraining concept or not. To
check whether a data type models a concept represented by algebraic specification, it must checked that the
concept map specification actualizes all the implicit parameters of the concept and defines axioms for all
the operations of the corresponding concept on the provided actual data type. Definition 3 specifies formally what it
means to say that a data type d with the specification D models a concept cp with the specification CP =
\( \lambda CTP.\{CIP\}.CSP \).

**Definition 2:**

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

\[ \text{opnam(SP)} = \{ f | f : s1 \times ... \times sn \rightarrow s \in \text{opns}(SP) \} \]

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

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

**Definition 3:**

A class d with transformation D is said to model a concept cp with transformation CP = \( \lambda CTP.\{CIP\}.CSP \) if
and only if the following three conditions are satisfied:

1. Condition 1:
   \[ \text{opns}(CM(CP[CTP/D])) - \text{opns}(CP[CTP/D][CIP/ACIP]) = \phi \]

2. Condition 2:
   \[ \text{opnam}(CP[CTP/D][CIP/ACIP]) - \text{axmnam}(CM(CP[CTP/D])) = \phi \]

3. Condition 3:
   \[ \text{imppar}(CM(CP[CTP/D])) = \phi \]
where CIP is the associated type in CP and ACIP is its implementation in CM(CP[CTP/D]).

That is, a type is said to model a concept if and only if all the associated types and all the operations of that concept is implemented by that type. This means that the concept map of that type must define all the associated types and 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 10:**

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:**

```cpp
template <typename TP> where ValueComparable<TP>
class GeometricProcesses {
    float GetAreaDifference(TP u, Rectangle v) {
        If (Compare(u,v)) {
            return(u.Area() - v.Area());
        } else {
            return(OutOfRange);
        }
    }
}
```

This class is transformed into a specification that when it is actualized with the actual data type, must be tested if it models the ValueComparable concept. If so then it can be instantiated. If it does not model the concept then it is an error specification that represent the failure is instantiated. The class is transformed into the specification:

```
GEOMETRICPROCESSES = λTP.CSP
CSP = import TP, VALUECOMPARABLE[CTP/TP]
sorts
TP
opns
GetAreaDifference: TP × TI → TP
vars
u: TP, v: TI
axms
GetAreaDifference(u,v)=(CompareValue(u,v)?u:v)
```

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

Assume that c is a constrained parameterized class, C = λP.CSP is its transformation, let AC be a Σ-C[P/ACTP] algebra, C is interpreted as follows:

**Interpretation Rule 5:**

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 6:**

If E is an element class that is used as a sort for parameters or attributes in C, let AE be a Σ-E then AC interprets each operation in opns (E) as in AE. that is:
∀_f_ : s_1 × ... × s_n \rightarrow s \in \text{opns}(F), \forall a \in E s_1 × ... × E s_n:
    f^{AC}(a) = f^{AE}(a)

**Interpretation Rule 7:**
If P is the parameter class that is used as a sort for parameters or attributes in C, let AP be a _Σ_-P then AC Interprets each operation in opns (P) as is in AP. that is:
∀_f_ : s_1 × ... × s_n \rightarrow s \in \text{opns}(P), \forall a \in P s_1 × ... × P s_n:
    f^{AC}(a) = f^{AP}(a)

**Interpretation Rule 8:**
Let cp be the constraining concept of the parameterized class, let ACP_ACTP be a _Σ_-CM(CP[CTP/ACTP]) algebra then AC Interprets each operation in opns (CM(CP[CTP/ACTP])) is as in AACTP. that is:
∀_f_ : s_1 × ... × s_n \rightarrow s \in \text{opns}(CM(CP[CTP/ACTP])),
    \forall a \in ACP_ACTP s_1 × ... × ACP_ACTP s_n:
    f^{AC}(a) = f^{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 = _λ_ P.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 _Σ_-PSC[P/ACT] algebra, _Σ_-ACT algebra respectively that results from the interpretation rules, then AC|_Σ_-ACT is a model of ACT.

**Proof:**
PSC, ACT are achieved by following the presented transformation rules and due to transformation rule 6 (part 5) sorts(P) contains a sort p. By transformation rule 7 (part 3) Since P is imported into SC then sorts(SC) also contain the sort p and sorts(P) \subseteq sorts(PSC), opns(P) \subseteq opns(PSC). From the definition of actualization PSC[P/ACT] replaces each occurrence of P by ACT. By transformation rule 10 part 3, PSC[P/ACT] replaces p by act. By interpretation rule 4 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 _Σ_-PSC[P/ACT] and contains the interpretation for all of the sorts and operations of ACT as interpreted in AACT, which means that AC|_Σ_-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.

**Lemma 3:**
Let c be a generic class p its parameter, cp its constraining concept on parameter p, let PSC = _λ_ P.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 _Σ_-PSC[P/ACTP] algebra, _Σ_-CM(CP[CTP/ACTP]) algebra respectively that results from the interpretation rules, then AC|_Σ_-CM(CP[CTP/ACTP]) is a model of CM(CP[CTP/ACTP]).

**Proof:**
By transformation rule 7 (part 5) sorts(P) contains a sort p. By transformation rule 7 (part 3) Since P is imported into SC then sorts(SC) also contain the sort p and sorts(P) \subseteq sorts(PSC), opns(P) \subseteq opns(PSC). From the definition of actualization PSC[P/ACTP] replaces each occurrence of P by ACT. By transformation rule 10 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 7 AC contains an interpretation for act. By interpretation rule 8 each operation in opns(CM(CP[CTP/ACTP])) is interpreted in AC. Since AC is a model for _Σ_-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|_Σ_-CM(CP[CTP/ACTP]) is a model of CM(CP[CTP/ACTP]).
Example 5:
In this example, the class Geometric Processes is instantiated using the parameter Circle (i.e Geometric Processes<Circle>), we give the interpretation of the specification GEOMETRICPROCESSES. We must first give the interpretation of the concept map for the constraining concept Value Comparable on the parameter Circle. The specification CM(VALUECOMPARABLE[T/CIRCLE]) is interpreted by the:

$$\sum_{CM(VALUECOMPARABLE[T/CIRCLE])}$$ algebra VALUECOMPARABLE_CIRCLE, usemin is interpreted arbitrarily by interpretation rule 3. The method Compare Value is interpreted in GEOMETRICPROCESSES as follows:

$$CompareValue^{VALUECOMPARABLE\_CIRCLE}(a, b) = (Area^{CIRCLE}(a) > Area^{RECTANGLE}(b))$$

The specification is interpreted by the $$\sum_{GEOMETRICPROCESSES}$$ algebra GEOMETRICPROCESSES, GeometricProcesses is interpreted arbitrarily by interpretation rule 3. The method GetAreaDifference is interpreted in GEOMETRICPROCESSES as follows:

$$GetAreaDifference^{GEOMETRICPROCESSES}(gp, a, b)$$
$$= if(CompareValue^{GEOMETRICPROCESSES}(a, b), Area^{Circle}(a), SPOUTOFRANGE)$$

$$CompareValue^{GEOMETRICPROCESSES}(a, b) = CompareValue^{VALUECOMPARABLE\_CIRCLE}(a, b)$$

Semantics of Object Binding and Embedding the Imperative Parts:
In this section, we describe how to integrate the semantics of the operation bodies which are of imperative nature with the axioms of the algebraic specifications. We give a semantic meaning of creating instances (or objects) of classes during compilation. To model assigning a value to an attribute, interpretations of attribute declarations have to be adapted to the values given.

Transformation of Bindings:
For representing binding of an object to a specific value, we define an assignment function. Let IDENT be the set for identifiers and VALUE be the set for values.

Definition 4:
An assignment is defined as a partial function from the set of all identifiers to the set of all values: assign:= IDENT → VALUE. The set of all assignments is denoted by ASSIGN.

Objects of primitive types can be assigned values directly. When a class is instantiated in the main function in the program an object of that class is created and its attributes can be given values. We can represent the binding by a function [_[_]] which maps object binding into assignments:

$$[_[_]] : BINDING \rightarrow ASSIGN,$$
$$B \in BINDING, var \in IDENT:$$

$$[[B]](var) := \begin{cases} 
value, & \text{if type(var) \in \{int, Boolean, float, char\}} 
\text{unbound}, & \text{else}
\end{cases}$$

We define a binary infix operation _\_ that extracts an identifier used in attribute declaration of a user-defined type. This identifier is called qualifying identifier if its type is a self-defined class. The attributes in this class are also assigned values. This is done recursively.

Let B be a binding and qual a qualifying identifier in B, then we define:

$$[_[_]] : BINDING \times IDENT \rightarrow BINDING,$$
$$If B \in BINDING, qual \in IDENT, var \in IDENT :$$

$$[[B][qual]](var) := value, if qual is a qualifying identifier and var is a primitive type$$
$$\_\_, else$$

The values fixed in a binding have to be typed correctly. In the following, attrType refers to the primitive types int, bool, ..etc.
**Example 6:**
The following CDLAT-program contains classes with only attributes (data members) since operations are not relevant to the subject of binding. The main function defines an object of the class Sponger.

```plaintext
class grower {
public:
    int num;
    //operations …
}

class sponger {
public:
    grower gw;
    int num;
    // operations …
}

t int main()
{
    sponger sp;
    int num;
    sp.gw.num = 5;
    sp.num = 7;
    num = 3;
    return 0;
}
```

The binding in the main function assigns values to the local `num` in the function and to the `num` in `sp` and to `num` in `gw` in `sp`. This binding is transformed into an assignment function. The binding of the local `num` yields ([B](num) |= 3), the `num` attribute in `sp` yields ([B|sp](num) |= 7), the `num` attribute in `gw` in `sp` yields ([B|sp|gw](num) |= 5). There is name clash for defining the attributes with the same name in different scopes. This is respected by introducing B|sp and B|sp|gw.

**Interpretation of Bindings:**
Using binding we can fix the interpretation of class attributes. Let c be a CDLAT-class, C its transformation, and AC a ΣC-algebra. Let e be an element class in c, E its transformation, and AE a ΣE-algebra. Let B be a binding that binds all attributes of c. Here, attrType refers to the primitive types int, bool, ….

**Interpretation Rule 1:**
For each attribute declaration in c of the form attrID: attrType we define:  
For each attrID : c → attrType ∈ opns(C), c ∈ AC : attrIDA(c) = [[B]](attrID)

**Interpretation Rule 2:**
For each attribute declaration in e of the format attrID: attrType we define:  
attrID : e → attrType ∈ opns(ELEM), e ∈ AE : attrIDE(e) = [[B|e]](attrID)

The interpretation of attribute declarations in element classes is forwarded to aggregates and by the interpretation rules which were given in the interpretation rules of classes. Consequently, a binding is respected in ΣE-algebra. The ΣC-algebra following the interpretation rules is called a bound model of C. The set of all bound models is denoted by Bound(C). Bound models are models, that is, all models that are in Bound(C) are also in Mod(C). This is because the interpretation rules fix the interpretation of attribute declarations to certain values of the respective carrier-sets which has no effect on maintaining the model properties.

**Example 7:**
Let S be a ΣSponger-algebra, and G a ΣGrower-algebra. We apply interpretation rules 1 and 2 for binding for the previous example resulting with the following interpretations:
1. With ([B|sp](num) |= 7) and rule 2, num⁵(s) = 7 for each s ∈ Ssponger by rule 1,
2. With ([B|sp|gw](num) |= 5) and rule 2, num⁵(g) = 5 for each g ∈ Ggrower by rule 2, num⁵(g) = num⁵(g) = 5.
Object Handling and Embedding Imperative Parts:
In CDLAT, imperative statements are used within function bodies. These statements include: declaration of variables, sequences, if statements, for- and while-loops, object creation and state updates. We stated in the transformation rules that the method bodies are transformed into suitable axioms. Here, we show how to integrate the denotational semantics of the imperative parts with the algebraic semantics.

Object Handling:
An object is an individually identifiable class instance. The state of an object is defined as the set of attribute/value pairs. This state can be changed using the methods which change the values of the attributes. Types are given through model sets, and each instance of a c corresponds to a model in Mod(C). To interpret objects, the type is extended to pairs of an identifiers and a model set: (IDENT, MOD), where an identifier correspond to the object name. Referencing is represented by a partial function ref: IDENT→MOD. Object modification is represented by modifying its corresponding model, m, by modifying axioms resulting in a new model m0. When the value of the object is changed, ref is also changed (ref (i) = m to ref (i) = m0). Deletion of an object, i, is represented by an undefined value for ref (i). A declaration of an object obj of type t is represented by adding a pair (obj, M) to ref where M has to be a model in Mod(T) (where T is the transformation of t), and obj is in IDENT.

Embedding Imperative Parts:
Within method bodies primitive statements and control structures are used such as declaration of variables, value assignment, method invocation, sequences, if statements, for- and while-loops. In order to transform method bodies into axioms, the semantics of the imperative parts has to be integrated with the algebraic semantics. We can use term evaluation to integrate denotational semantics of imperative languages for integrating the imperative parts. When CDLAT-program is transformed and an interpreted correctly, program execution can be viewed as the evaluation of its interpretation. Let us first define term evaluation.

Definition 5:
A mapping v : X → A is called valuation. It is defined throughan S-indexed family {vs}s∈S of mappings vs : Xs→As for each s ∈S. Let Σ= <S,Γ> be a signature, A aΣ-algebra, and v : X →A a valuation. For each s ∈S, a S-sorted mapping vs*:Σ (X)s→As is defined as follows:
1. for all variables x ∈Xs let v*s (x) := v(x)
2. for all constants f : s→s∈Γ let v*s (f) := f^v
3. for all operations f : s1×...×sn→s∈Γ and all terms t1 ∈TΣ (X)s1, . . . , tn ∈TΣ (X)sn let vs* (f(t1, . . . , tn)) := f^v(vs* (t1), . . . , vs* (tn))

A mapping v*:Σ (X)→A is called term evaluation of terms in A with respect to v, and is defined through an S-indexed family {vs}s∈S of mappings vs*.

The execution of a CDLAT-program can be interpreted using the mechanisms of the imperative languages. The primitive statements are the assignments and method invocation. Controlling the order of execution is reduced to the interpretation evaluation. The program starts at the main function. Using the transformation rules, semantic functions, [[ ]], can be defined to denote a mathematical object to each syntactic construct. For example, the semantic function for a program, p, maps p onto the model set Mod(P) which results from interpretation of the transformation P. The declaration of a local object, i, of type t is given a meaning through updating the function that maps the object name to its type model, (i.e. by adding the pair (i, m) to ref, where m is in Mod(T), where T is the transformation of t. Also, we can formalize the imperative constructs in the method bodies algebraically using the methodology of denotational semantics. As an example, consider the denotational semantics of an if-statement of the form [[if b then s1 else s2]] which can be shown as follows:

[[if b then s1 else s2]] = \begin{cases} [[s1]], & \text{if } [[b]] = \text{true} \\ [[s2]], & \text{else} \end{cases}

To express this algebraically, we evaluate the semantics of s1, if the interpretation of b in a ZBOOL-algebra B is reduced to true. If it reduces to false, we evaluate s2. The ternary operation if_then_else_ : bool×s×s→s is given for each sort s ∈S with the semantics

if true then x else y =s x and if false then x else y =s y.

Consequently we can evaluate the denotational semantics of the if statement algebraically as follows:

[[if b then s1 else s2]] := v*(if b then s1 else s2)
This is how an if-statement is transformed. The semantic functions \([s1]\) and \([s2]\) are evaluated using the term evaluation \(v^*\) by reducing the mapping \([[]]\). For example, a member function call of the form \(e.m(p_1, \ldots, p_n)\) is transformed into a term \(\text{ident}(e, p_1, \ldots, p_n)\). The semantics of this method invocation is given by the evaluation:

\[
[[e.m(p_1, \ldots, p_n)]] := v^*(m([[e]], [[p_1]], \ldots, [[p_n]])) = m^k(v^*(e), v^*(p_1), \ldots, v^*(p_n)),
\]

where \(e\) is an instance of class \(E\), and \(E\) is a \(\Sigma_e\)-algebra. The semantics of variable declaration, sequences, and loops can be given in the same way. The semantics of a program is given by recursive transformation and evaluation which means that this algebraic semantics is denotational.

**Correctness of the Model:**

In this section we show the correctness of our model in describing the constrained template classes. The correctness can be proved by showing that the constrained class represented by the model cannot be instantiated over an actual type unless this type models the concept which constrained the generic class. In the following lemma, we prove that if the concept map of an actual type does not implement all the operations and the associated types of that concept then the instantiation of the class over that actual type fails.

**Lemma 4:**

Let \(c\) be a generic class \(p\) its parameter, \(cp\) its constraining concept on parameter \(p\) which requires \(n\) operations \(f_1, \ldots, f_n\) and an associated type \(cip\), \(CP\) its transformation, let \(C\) be the transformation of the class, let \(actp\) be an actual class, \(ACTP\) its transformations, let \(cp<actp>\) be the concept map of \(cp\) over \(ACTP\) which implements all the operations of the concept but does not provide a type definition for the actual class \(ACTP\). \(CM(CP[ACTP])\) its transformation of the concept map, \(PSC\), \(CM(CP[ACTP])\) are achieved by applying the transformation rules. Then, an object of the type \(c<actp>\) cannot be instantiated.

**Proof:**

By transformation rule \(3\) since each operation in \(cp\) has a transformation in \(CP\) then all the operations \(f_1, \ldots, f_n\) of \(cp\) are contained in \(CP\). By transformation rule \(2\), the associated type of the form \(cip\) is transformed into an implicit parameter \(CIP\) and added to the set of the implicit parameters. \(CIP\) is imported to \(CSP\). By transformation rule \(6\) part \(1\), a method signature in the concept map \(cp<actp>\) of the form \(mt \: mv(t_1 \times \ldots \times t_n \rightarrow m_t)\) is transformed into an operation in \(opns(CM(CP[ACTP]))\) of the form \(mv: t_1 \times \ldots \times t_n \rightarrow m_t\). By the asumption of the lemma all the operations are implemented by \(CM(CP[ACTP])\) which means that:

\[
opns(CM(CP[ACTP])) = \phi
\]

and

\[
opnam(CP[ACTP]) = \phi
\]

So the first two conditions of definition \(3\) are satisfied.

The third condition is not satisfied because the set of the implicit parameters of \(CM(CP[ACTP])\) is not empty since the typedef section is not provided and so the implicit parameter is not removed. That is:

\[
\text{imppar}(CM(CP[ACTP])) \neq \phi
\]

So, \(ACTP\) does not model the concept \(CP\). Recall from that the constrained class is transformed into a conditional specification of the form:

\[
C = \begin{cases} 
  AP, SP, & \text{if } P \text{ models } CP \\
  SPERROR, & \text{otherwise}
\end{cases}
\]

Due to the condition \(if \ P \text{ models } CP\) which is not true when \(P\) is replaced by \(ACTP\), so the instantiation \(c<actp>\) is transformed into the error specification \(SPERROR\) which represents the failure in an instantiation of object of the type \(c<actp>.\)

By the same approach, we can prove that: if the concept map \(CM(CP[ACTP])\) does not provide an implementation of an operation \(f_i\) of the concept \(CP\) then the instantiation of any object of type \(c<actp>\) will fail.

In the following lemma, we prove that if the concept map of an actual type implements all the operations and the associated types of that concept then the instantiation of the class over that actual type succeeds.
Lemma 5:
Let \( c \) be a generic class \( p \) its parameter, \( cp \) its constraining concept on parameter \( p \) which requires \( n \) operations \( f1, \ldots, fn \) and an associated type \( cip \), \( CP \) its transformation, let \( C \) be the transformation of the class, let \( acctp \) be an actual class, ACTP its transformations, let \( cp<acctp> \) be the concept map of \( cp \) over ACTP which implements all the operations of the concept and provide a type definition for to actualize the associated type, CM\([CP][(CTP/ACTP)]\) its transformation of the concept map, PSC, CM\([CP][(CTP/ACTP)]\) are achieved by applying the transformation rules. Then, an object of the type \( c<acctp> \) can be instantiated.

Proof:
By transformation rule 3 since each operation in \( cp \) has a transformation in \( CP \) then all the operations \( f1, \ldots, fn \) of \( cp \) are contained in \( CP \). By transformation rule 2, the associated type of the form \( cip \) is transformed into an implicit parameter \( CIP \) and added to the set of the implicit parameters. CIP is imported to CSP. By transformation rule 6 part 1, a method signature in the concept map \( cp<acctp> \) of the form \( mt \) \( (t1 \times \ldots \times tn \Rightarrow mt) \). By the assumption of the lemma all the operations are implemented by CM\([CP][(CTP/ACTP)]\) which means that:

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

and

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

So the first two conditions of definition 3 are satisfied.

The third condition is also satisfied because the set of the implicit parameters of CM\([CP][(CTP/ACTP)]\) is empty since the typedef section is provided and so the implicit parameter is removed. That is:

\[\text{impmap}(CM(CP[CTP/ACTP])) = \phi\]

So, ACTP models the concept \( CP \). Recall from that the constrained class is transformed into a conditional specification of the form:

\[C = \begin{cases} \lambda P.SP, & \text{if } P \text{ models } CP \\ \text{SPError}, & \text{otherwise} \end{cases}\]

Due to the condition (\( if \ P \text{ models } CP \)) which is true when \( P \) is replaced by ACTP, so the instantiation \( c<acctp> \) is transformed into the parameterized specification \( \lambda P.SP \) which represents the instantiation of the type \( c<acctp> \).

Conclusion:
In this paper, we defined a mini-language CDLAT such that its concept can define one parameter and one associated type. Associated types are considered as implicit parameters. They are like place holders for types that need to be bound to an actual type. We gave the rules for transforming the concepts of CDLAT such that every concept is transformed into a parameterized algebraic specification that contains one explicit parameter and one implicit parameter. The explicit parameter represents the direct parameter of the concept. The implicit parameter represents the associated type. The function signatures are represented as operations in the specification. A formal description of the condition of type-concept-modeling was needed to include the associated type.

For the concept map, we gave rules to transform the concept map into a simple specification over the actual type. The type definition section is represented by replacing all the occurrences of the associated type with the actual type. We gave a proof to that the semantics of the concept map preserves the semantics of the both the explicit and implicit parameters of the corresponding concept. Also, our model emphasizes on the property that in order to that an actual type model a concept, it must implement both its operations and its associated types. Representing associated types is an important aspect of concepts, since they represent type mapping in a form of parameters and abstract types. Also, the direct representation of the encapsulated associated types makes it easier to use. This can be considered as a step in defining the formal semantics for the newly suggested construct (the concept) for generic programming in C++.

Our approach emphasizes that the concepts that are object-oriented structures present a mechanism for structuring that are not available in imperative languages. It emphasizes on the fact that concepts are structures giving constraints on the generic parameters. For this we used parameterized specifications for concepts which are closer to the template class structure than the flat specification approach that is used by other approaches. Also, our approach is distinguished by using the usual parameterized specification generic classes and using the parameterized specification with explicit and implicit parameters to transform the concepts. This approach directly reflects the nature of the concepts that defines a generic type parameter and an associated type.
The methodology used in this paper can be used to define transformation rules for concepts with multiple parameters and multiple associated types. As a future work, we plan to use the algebraic specification technique for the description of concept refinement.

REFERENCES


