Constraint Verification of Generic Algorithmic Program for Solving General Network Path Problems

Zhengkang Zuo
National Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
Provincial Key Laboratory for High-Performance Computing Technology, Jiangxi Normal University, Nanchang, China
Graduate University, Chinese Academy of Sciences, Beijing, China
Email: kerrykaren@126.com

Jinyun Xue, and Changjing Wang
Provincial Key Laboratory for High-Performance Computing Technology, Jiangxi Normal University, Nanchang, China
Email: jinyun@jxnu.edu.cn, wcj771006@163.com

Abstract—Generic programming has emerged as a paradigm for the development of highly reusable and safe software libraries. Generic constraint mechanism can detect and verify the validity of generic parameter instantiated, thereby guarantee dependability and safety of generic programming. Kleene algorithm is ingenious for solving general network path problems but there is a lack of research concerning how to verify the closed semi-ring constraint of the algorithm. First, the paper describes the related works of generic constraints. Next, based on a new description of generic constraints of Apla language by means of algebraic structures, presents the case of generic Kleene algorithm, which is hard to describe and verify its constraint by the current method. Then, with the help of Isabelle theorem prover, verifies closed semi-ring constraint of Kleene algorithm. Finally, transforms the Apla programs of generic Kleene algorithm to C++ codes automatically based on C++ generator. Success in this example gives us more evidence that the approach does not only support for appropriate semantic constraints, but also markedly improves dependability and safety of generic programming.

Index Terms—generic programming, constraint verification, closed semi-ring, Kleene algorithm, algebraic structures

I. INTRODUCTION

It is a challenging task of computer scientists for increasing the efficiency of developing software. For answering the challenge, we are developing the PAR method and its supporting platform [1], called PAR platform that is a long-term research project supported by a series of research foundations of China. PAR method and PAR platform consist of PAR programming methodology, specification and algorithm describing language Radl [2], abstract programming language Apla, a set of rules for specification transformation [3,4] and a set of automatic transformation tools of algorithm and program. PAR provides powerful generic structures that support convenient generic programming and the methodology that supports formal derivation of executable algorithmic programs from their formal specification.

Pioneered by Alexander Stepanov and David Musser [5], generic programming (GP) is a programming paradigm for developing efficient, reusable software libraries [6]. Existing programming languages, i.e. C++ and Java, which support generic mechanism, describe constraints of the data type parameter solely. However, it is low level of abstraction and hard to formal verification, so limits the application of generic programming.

Network path problems are classical ones in a directed graph. Several famous graph algorithms related with them. Typical examples are Warshall’s algorithm for computing the transitive closure of a graph, Floyd’s algorithm for computing the all-pairs shortest path, etc. Aho, Hopcroft and Ullman [7] described a generic algorithm using an algebra structure called a closed semi-ring, originally derived from the work of Kleene [8], for solving general network path problems. In this paper, we name the algorithm as Kleene algorithm. Backhouse’s paper [9] derived Floyd and Dijkstra’s path algorithm based on algebra of regular languages. Gries presented formal description of Warshall’s graph transitive algorithm in [10]. J. Y. Xue in [11] attempted to derive Kleene algorithm for solving several network path problems just using a simple and practicable approach.

Kleene algorithm is ingenious for solving general network path problems but there is a lack of research concerning how to verify the closed semi-ring constraint of the algorithm. Thereby, there is no guarantee of dependability and safety for the generic algorithm program. Current studies of generic constraint only support for syntax requirements, but not for semantic requirements. Therefore, it is hard to describe and verify the constraint by the current method.
This paper illustrates the related works of generic constraints. Presents a new description of generic constraints of Apla language by means of algebraic structures, then describes one typical case-generic Kleene algorithm in Apla language, and verifies closed semi-ring constraint with the help of Isabelle theorem prover. Finally uses the Apla to C++ generator to realize generic Kleene algorithm and its constraint mechanism automatically.

The organization of this paper is as follows. In Section II, we describe the related works of generic constraints. In Section III, we present the case of generic Kleene algorithm for solving general network path problems in Apla. Closed semi-ring constraint of Kleene algorithm is verified in Section IV. In Section V, we transform the Apla programs of generic Kleene algorithm to C++ codes based on C++ generator automatically. Finally, we draw conclusions and figure out future work in Section VI.

II. RELATED WORKS

Musser presented recognized definition of generic constraint [5].

Definition 1. Generic constraint
Given A is a set of abstractions (such as Abstract Data Type (ADT)), and generic constraint C is a set of requirements R. That is:

\[ C(R/A) = \{ a \in A | \forall r \in R, r(a) = \text{true} \} \]

In which, A is a set of all types in the domain, and R is commonness of all elements in C.

Current work of generic constraint includes C++ templates, concepts of ConceptC++, generics of C# and Java and named type constraint proposed by Swen Bing, professor of Peking University.

A. C++ Templates
C++ templates are the core of the design of current successful mainstream libraries and systems [12]. However, they lack formal descriptions of generic constraint. The near-optimal performance offered by ISO C++ templates comes at the price of very weak separation between templates definitions and their uses. It may lead to huge practical problems, such as obscuration of generic concept, nonsupport of instant type check and indigestibility of compiling information [13].

For instance, Forward Iterator is generic concept cited from STL. Table I shows its part requirements.

There is the example of the standard template function `fill`, which conforms to ANSI/ISO C++ standard, shown in Fig. 1. TABLE I shows parts requirements of generic concept “Forward Iterator”.

However, in the Fig. 1, there is no explicit description of the `Forward Iterator` requirements. Thus, compile of C++ would not check whether type parameter `Iter` conforms to generic concept `Forward Iterator` or not. For example, in the line 7 of the Fig. 1, we adds one expression `first-`. Although the expression does not conform to the `Forward Iterator` constraint, compile does not point out errors in the place of definition of function `fill`.

B. Concepts of ConceptC++

Compare to C++ templates, concepts of ConceptC++ generate sets of "primitive operations" with signatures from a simpler and more abstract language and adds formal description of generic constraints explicitly [14].

Definition 2. Syntax description of concept
Concept `ConceptName <P> where G {B}`

It is a triple `<P, G, B>` where:
- P is a list of explicit concept-parameters, with exactly the same declaration syntax as same for template parameters.
- G is the guard. It is a logical formula, made of compile time expressions combined with the usual logical operators. The where-clause is optional. It usually expresses additional assumptions on combinations of the parameters P.
- B is the body of the concept. It is a sequence of simple declarations and expression-statements that enunciate syntax and type equations between the concept-parameters.

ConceptC++ supports for syntax requirements and instant type check. However, it does not support for semantic requirements. Furthermore, because of complexity of using concepts, ConceptC++ was dramatically voted out of C++0x during the C++ standards committee meeting in Frankfurt in July 2009.

C. C#/Java Generics

Compare to C++ templates, C#/Java generics can provide enhanced safety but are also somewhat limited in capabilities. They introduce “F-Bounded Parametric Polymorphism” to constrain type parameters [15]. Type variables in a parameterized type can be bounded by a class or an interface, and then only subtypes of the bound can be used to instantiate the parameterized type.

As inheritance mechanism is suitable for implement “F-Bounded Parametric Polymorphism” [16], C#/Java generics can be realized easily. Nevertheless, C#/Java generics are considered narrow constraints at best.
Because of using “F-Bounded Parametric Polymorphism”, parameterized type should be instantiated by reference data types, not by elementary data types [17].

D. Named Type Constraint

Swen Bing, professor of Peking University, proposed named type constraint mechanism; selected C++ language as the host language; designed the standard constraints library; developed a compiler front-end (has been integrated into the C++), as far as possible to reuse existing C++ resource [5]. It is overly dependent on C++.

Because abstract level of the generic mechanism is relatively low, it is difficult to achieve high level of abstraction of generic programming requirements. Second, it does not contain semantic constraints requirements.

III. GENERIC KLEENE ALGORITHM

A. Apla Generic Constraint Mechanism

Apla is abbreviation of Abstract Program Language [1]. It is a part of PAR method and is an object-based abstract programming language with convenient generics. The purpose of developing Apla is to implement functional abstract and data abstract perfectly in the development of programming, so that any Apla program is simple enough and is ease for understanding, formal derivation or proof [18]. It is also easy to transform into some OOP language programs, such as C++, Java and C#, etc. The language absorbs some control structures from Dijkstra’s Guarded Command Language, but restricts the non-determinacy. Apla is generic programming language. It supports data value, data type and subroutine as parameter of ADT, procedure and function.

In this paper, we add generic constraints mechanisms into Apla language. The mechanism focuses on algebraic structures semantic constraint. It includes three parts: constraint definition, constraint call and constraint instantiation. Based on constraint definition, we design a predefined Apla algebraic structur es generic constraint. Then users can constrain that parameterized types and subroutines should conform to constraints requirements in the step of constraint call. At last, constraint instantiation is instantiating generic types into specific types based on constraints. BNF description of Apla generic constraints mechanism is described in [19].

Abstract data type defines a collection of data and a set of operations on the data set. It can be viewed as corresponding to a series of elements and their algebraic operations. Operational semantics can be described by using algebraic equation axioms. As the different operations and equation axiom can form all kinds of algebraic system specification, any ADT can be described by algebraic system specification. We proposed to construct the appropriate algebraic structure, which unifies a class of problems in a mathematical model or a library of reusable programming components.

In this paper, we present the algebraic system specification language. Each Theory consists of three parts: syntax definition, semantic definition and Theory importing.

Definition 3. Algebraic system specification language
Theory theory_name;
sorts: type_name_list;
opers: operation_name: type_name_list-> type_name;
......
where: theory_importing_list
eqns: for quantifier variable_declaration
equation_left=equation_right
......
EndTheory;

In the part of syntax definition (sorts and operes), type_name_list are type names of the operating domain, and type_name is type name of operating result. In the part of semantic definition (eqns), algebraic equation is adopted to depict the semantics of the operation. Equation_right also allows conditional expressions and recursion.

Example 1. Algebraic system specification of semi-ring
Theory Semi-ring;
sorts: item;
opers: ⊕: item×item → item;
⊗ : item×item → item;
where: Abelian- monoid (elem, ⊕ ) ∧ Monoid (elem, ⊗);
eqns: for ∀ x,y,z: item;
x ⊗ (y ⊕ z) = (x ⊗ y) ⊕ (x ⊗ z) ∧ (y ⊕ z)
⊕ x = (y ⊗ x) ⊕ (z ⊗ x)
EndTheory;

Based on algebraic system specification of semi-ring, which is presented in Example 3, combined with Apla generic constraint mechanism, we present constraint definition of semi-ring in Apla.

Example 2. Constraint definition of semi-ring in Apla
define constraint Semi-ring;
define ADT T (sometype elem);
someop ⊗ (a,b; elem) : elem;
someop ⊗ (a,b; elem) : elem;
enddef;
generic <someADT T> where (Abelian- monoid (T (elem, ⊕ )) ∧ Monoid (T (elem, ⊗ )));
∀(x, y, z: T, T, T = elem : x ⊗ (y ⊗ z) = (x ⊗ y)
⊕ (x ⊗ z) ∧ (y ⊗ z) ⊗ x = (y ⊗ x) ⊕ (z ⊗ x));
enddef;

According to the Apla generic constraints mechanisms, we have designed a library of constraints based on algebraic structures. It includes Binary-op, Left-identity, Right-identity, Identity, Commutative, Groupoid, Semigroup, Monoid, Abelian-monoid, Left-inverses, Right-inverses, Inverses, Group, Abelian-group, Ring, Semiring etc. Compared with the related works, the method presented in this paper improves the abstract degree of generic constraint, and supports semantic constraint. Thus, it facilitates formal verifying.

B. Generic Kleene Algorithm by Apla

Closed semi-ring has applications in various branches of computing such as automata theory, the theory of grammars, the theory of recursion and fixed points,
sequential machines, aspects of matrix manipulation, and various problems involving graphs, e.g. finding shortest-path algorithms within graphs. Kleene algorithm unifies a family of network path problems, including Floyd’s all-pairs shortest path algorithmic program, Washell’s graph transitive closure algorithmic program and maximum capacity path algorithmic program, defined on directed or undirected graphs [7]. It can be described as following:

• Shortest path problem. The closed semi-ring \((1 \geq 0 \cup \infty, \min, +, \infty, 0)\) corresponds to the shortest path problem. The shortest path problem is to be defined on a set of nonnegative integer numbers, \(1 \geq 0 \cup \infty\). The labeling function returns the cost of the path, \(\lambda(i, j) = c_{i,j}\). The extension operator is the arithmetic operator +. The summary operator is the min operator.

• Transitive closure problem. The closed semi-ring \((\{0, 1\}, \lor, \land, 0, 1)\) corresponds to the transitive closure problem. The labeling function \(\lambda(i, j)\) returns 1 if there is an arc between i and j, otherwise it returns 0. The summary operator is the logical OR operator while the extension operator is the logical AND operator.

• Maximum capacity path problem. The closed semi-ring \((1 \geq 0 \cup \infty, \min, \max, \infty, 0)\) corresponds to maximum capacity path problem. The extension operator is the max operator while the summary operator is the max operator.

Apla description of generic Kleene algorithm includes three parts: constraint definition, constraint call and constraint instantiation. It can be described as following:

1) Constraint definition.

**Definition 4.** A closed semi-ring \(<S, \otimes, \theta, I, \triangleright>\) is a set \(S\), equipped with two binary operations \(\otimes\) and \(\theta\), called addition and multiplication, and two constant elements \(I\) and \(\triangleright\), such that:

1. \(<S, \otimes, \theta>\) is a commutative monoid with identity element \(\triangleright\), that is for all \(a, b, c \in S\):
   - \((a \otimes b) \otimes c = a \otimes (b \otimes c)\)
   - \(a \otimes b = b \otimes a\)
   - \(a \otimes \triangleright = \triangleright \otimes a = a\)

2. \(<S, \triangleright, \triangleright>\) is a monoid identity element \(I\), that is for all \(a, b, c \in S\):
   - \((a \otimes b) \otimes c = a \otimes (b \otimes c)\)
   - \(a \otimes I = I \otimes a = a\)

3. Multiplication distributes over addition, that is for all \(a, b, c \in S\):
   - \((a \otimes (b \otimes c)) = ((a \otimes b) \otimes c)\)
   - \(c \otimes (a \otimes b) = (c \otimes a) \otimes (c \otimes b)\)

4. \(\theta\) annihilates \(S\), that is for all \(a \in S\):
   - \(a \otimes \theta = \theta \otimes a = \theta\)

5. Multiplication distributes over countably infinite addition, that is:
   \[
   \sum_{i=0}^{\infty} a_i = a_1 \otimes a_2 \otimes a_3 \ldots = \sum_{i=0}^{\infty} a_i \otimes b_i
   \]

Based on the Definition 4, we give Apla constraint definition of closed semi-ring, that is:

```apl
define constraint Closed-semiring;
define ADT T (sometype elem);
someop @ (a, b: elem): elem;
someop @ (a, b: elem): elem;
enddef;
```

```
generic <someADT T> where (Basetype (T, elem) \land Basebinaryop (T, @) \land Basebinaryop (T, @) \land (Semi-ring (T elem, @, @)));

\forall (x, y: T) x, y = integer; a[x], a[y] = elem:

(\sum_{x=0}^{\infty} a[x] \otimes (\sum_{y=0}^{\infty} b[y]) = (\sum_{x=0}^{\infty} (a[x] \otimes b[y]));
```

2) Constraint call.

[11] presents a formal derivation of a generic algorithmic program for solving general network path problem using PAR method. The generic algorithm described in Apla is presented as following. This procedure calls the custom Closed-semiring constraint, which is defined in the step of constraint definition.

```
generic <someADT T>;
procedure Kleene (n: integer; d: array [1..num, 1..num, elem]) where (Closed-semiring (T (elem, @)));
//called Closed-semiring constraint
```

**3) Constraint instantiation.**

The Kleene generic algorithm unifies a series of network path algorithms of graph, including the shortest path algorithm, the transitive closure algorithm and the maximum capacity algorithm. If we select appropriate closed semi-ring structure by instantiation statement replacing the ADT parameter T in the generic procedure, we can generate the specific algorithm to solve the different problem.

For example, selecting closed semi-ring structure \((I \cup \infty, \min, +, \infty, 0)\) and executing instantiate statement, like that:

```
ADT A1: new T (integer; min; +);
```

procedure floyd: new Kleene (instantiation Closed-semiring (A1));

© 2013 ACADEMY PUBLISHER
In addition, selecting closed semi-ring structure \(\{0, 1\}, \lor, \land\) and executing instantiate statement, like that:

**ADT A2:** new T (boolean; \lor; \land);

procedure close-set: new Kleene (instantiation Closed-semiring (A2);

In addition, selecting closed semi-ring structure \((\Gamma \cup \infty, \min, \max, \times, 0)\) and executing instantiate statement, like that:

**ADT A3:** new T (integer; max; min);

procedure capacity: new Kleene (instantiation Closed-semiring (A3);

We can generate shortest path algorithm, transitive closure algorithm and maximum capacity algorithm respectively.

**IV. CLOSED SEMI-RING CONSTRAINT VERIFICATION**

Constraints matching are divided into two parts: constraints matching detection and constraints matching verification. Constraints matching detection is supposed to determine whether formal parameters and instantiation parameters satisfy syntax requirements of constraints. The process can be accomplished automatically based on the PAR platform. Constraints matching verification is supposed to determine whether the instantiation parameters satisfy the semantics requirements of constraints. This process is partial automation [20]. It needs to deduce verifiable instantiation terminal expansion manually, and then verify its correctness with the help of Isabelle theorem prover [21].

As constraints matching detection is supposed to be accomplished automatically based on the PAR platform, detection process of the Kleene generic algorithm will be shown in Section V. This section presents constraints matching verification process of the algorithm. Steps are shown as the following:

**Step 1.** Replace generic parameter T in the constraint definition of the closed semi-ring with instantiated abstract data types: (integer;min;+), (boolean; \lor; \land) and (integer;max;min).

**Step 2.** The constraint definition of the closed semi-ring is unfolded into constraint instantiated expansion 1, 2, 3.

- **Instantiated expansion 1:**
  
The first includes constraint instantiation refinements R1:
  
  \{Basetype (integer), Basebinaryop (min), Basebinaryop (+), \}
  
  Semi-ring (integer;min;+) and instantiation terminal expansion A1:
  
  \(\forall(x, y: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: a(x), a(y) = integer: (\sum_{x=0}^{\infty} a(x) + \sum_{y=0}^{\infty} b(y))\);

- **Instantiated expansion 2:**
  
The first includes constraint instantiation refinements R2:
  
  \{Basetype (boolean), Basebinaryop (\lor), Basebinaryop (\land), \}
  
  Semi-ring (boolean; \lor; \land) and instantiation terminal expansion A2:
  
  \(\forall(x, y: \exists, \mathcal{P}, \exists, \mathcal{Z} = boolean: a(x), a(y) = boolean: (\sum_{x=0}^{\infty} a(x) \lor \sum_{y=0}^{\infty} b(y))\);

**Step 3.** Unfold constraint instantiation refinements R1, R2 and R3 respectively until all expansions transform to instantiation terminal expansions. The paper chooses ADT (integer;min;+) as an example to illustrate the unfolding process.

**Step 4.** Replace generic parameter T in Closed-semiring constraint with ADT (integer;min;+), we get four constraint instantiation refinements: Semi-ring (integer;min;+), Basebinaryop (min), Basebinaryop (+) and one instantiation terminal expansion A1. Then unfolding Semi-ring (integer;min;+), we can get eight instantiation terminal expansion B1-B8. It is shown in the Fig. 2.

Based on definition of algebraic structure, we present instantiation terminal expansion B1-B8.

B1: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: x + (\min(y, z)) = \min((x + y), (x + z))\);

B2: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: \min(\min(x, y), z) = \min(x, \min(y, z))\);

B3: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: \min(x, z)\);

B4: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: \min(x, y)\);

B5: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: x + y\)

B6: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: x + y\)

B7: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: x + y\)

B8: \(\forall(x, y, z: \exists, \mathcal{P}, \exists, \mathcal{Z} = integer: x + y\)

**Step 5.** Unfold Basebinaryop (boolean), Basebinaryop (min) and Basebinaryop (+). The process much simpler than unfolding Semi-ring (integer;min;+), so omitted.

**Step 6.** At present, constraint instantiated expansion 1 has been unfolded into instantiation terminal expansions B1-B8 and A1. Transform these instantiation terminal expansions into Isar language scripts [21]. With Isabelle theorem prover, we have verified that instantiation terminal expansions B1-B8 and A1 are established. Thus, constraint instantiated expansion 1 is proved. It can be concluded that ADT (integer;min;+) conforms to constraint definition of close semi-ring. Due to limited space, we present part Isabelle theorem proving process of A1, as is shown in Fig. 3.

**Step 7.** As algebraic system A<1, min, \lor, \land> and ADT (integer;min;+) are isomorphic, through the verification process steps 1 to 6, it can be concluded that algebraic system A<1, min, \lor, \land> conforms to constraint definition of closed-semiring.
Step 8. Similar, it can be concluded that algebraic system 
\( C<1,\text{max},\min> \) conforms to constraint definition of closed semi-ring.

Step 9. There is homomorphic mapping relationship \( f: I \rightarrow \{0, 1\} \) between algebraic system \( B<\{0, 1\}; \lor; \land> \) and algebraic system \( C<1, \text{max}, \min> \), that is:
\[
    f(n) = \begin{cases} 
        1 & \text{if } n \geq 0, n \in I \\
        0 & \text{if } n < 0, n \in I
    \end{cases}
\]

Thereby, it can be concluded that algebraic system 
\( B<\{0, 1\}; \lor; \land> \) conforms to constraint definition of closed semi-ring according to the theorem 1.

Theorem 1. Given \( f \) is a homomorphism mapping of algebra system \( <A, +, *> \) to the algebraic system \( <B, \oplus, \otimes> \). If \( <A, +, *> \) is closed semi-ring and \( <B, \oplus, \otimes> \) has the homomorphic image of homomorphic mapping \( f \), then \( <B, \oplus, \otimes> \) is also a closed semi-ring.

Prove:

As \( <A, +, *> \) is closed semi-ring, \( <A, +, *> \) is monoid, obviously \( <B, \oplus > \) is Abel monoid and \( <B, \otimes > \) is monoid.

For all \( b_1, b_2, b_3 \in B \), existing the corresponding \( a_1, a_2, a_3 \), make 
\[
    f(a_1) = b_1 (i = 1, 2, 3),
\]
so
\[
    b_1 \oplus (b_2 \oplus b_3) = f(a_1) \otimes (f(a_2) \otimes f(a_3))
\]
\[
    = f(a_1) \otimes (f(a_2 + a_3))
\]
\[
    = f(a_1 \ast (a_2 + a_3))
\]
\[
    = f((a_1 \ast a_2) + (a_1 \ast a_3))
\]
\[
    = f(a_1 \ast a_2) \oplus f(a_1 \ast a_3)
\]
\[
    = (f(a_1) \otimes f(a_2)) \oplus (f(a_1) \otimes f(a_3))
\]
\[
    = (b_1 \otimes b_2) \oplus (b_1 \otimes b_3)
\]
In the same way, we can prove
\[
    (b_2 \oplus b_3) \otimes b_1 = (b_2 \oplus b_1) \otimes (b_2 \oplus b_1).
\]
So, \( <B, \oplus, \otimes > \) is semi-ring.

For all \( x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \in B \), existing the corresponding \( a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m \in A \) makes
\[
    f(a_i) = x_i \quad (i = \ldots, 1, 2, 3\ldots\infty)
\]
\[
    f(b_j) = y_j \quad (j = \ldots, 1, 2, 3\ldots\infty)
\]
It means:
\[
    \sum_{i=0}^{\infty} (a[i] \otimes (\otimes) f(j)) = \sum_{i=0}^{\infty} (f(a[i]) \otimes f(b[j]))
\]
\[
    = (f(a[0]) \otimes f(a[1]) \otimes \ldots f(a[\infty]))
\]
\[
    \otimes (f(b[0]) \otimes f(b[1]) \otimes \ldots f(b[\infty]))
\]
\[
    = (f(a[0]) + a[1] + \ldots a[\infty]) \otimes f(b[0] + b[1] + \ldots b[\infty])
\]
\[
    = f((a[0] + a[1] + \ldots a[\infty]) \ast (b[0] + b[1] + \ldots b[\infty]))
\]
\[
    = f(\sum_{i=0}^{\infty} (a[i] \ast b[j])) = \sum_{i=0}^{\infty} (x[i] \otimes y[j])
\]
So, \( <B, \oplus, \otimes > \) is closed semi-ring.

Step 10. It can be concluded that algebraic system \( A<1, \text{min}, \lor, >, B<\{0, 1\}; \lor; \land> \) and \( C<1, \text{max}, \min> \) all conform to constraint definition of closed semi-ring.
Apla programs are simple enough and ease for formal derivation and proof. However, they cannot be executed in a computer. Therefore, we developed the PAR platform that consists of six automatic generator tools of algorithms or programs [1]. C++ generator is one of PAR platform. There are two main functions: syntax check to Apla programs and generation of C++ programs automatically. Its system structure is shown in Fig. 4. As to generic programs, constraints matching detection is most important part of syntax check. It is supposed to determine whether formal parameters and instantiation parameters satisfy syntax requirements of constraints. The section selects generic Kleene algorithm as a typical example. In this case, C++ generator has accomplished three functions automatically. They are formal parameters detection, instantiation parameters detection and generation of C++ programs.

A. Formal Parameters Detection

Step 1. Based on constraint definition of closed semiring, automatically generate operational collection of type parameter elem: \( \{ \text{integer}, \text{real}, \text{char}, \text{boolean} \} \).

Step 2. Scan the Kleene generic procedure, then generate dependency expressions with elem: \( \text{write}(c[i, j]) \text{"","}\) and \( c[i, j] \oplus c[i,k] \otimes c[k,j] \).

Step 3. Generate related operations of dependency expressions: \( \otimes \) and \( \oplus \).

Step 4. As \( \oplus \) and \( \otimes \) belong to operational collection of type parameter elem, there is no error in formal parameters detection.

B. Instantiation Parameters Detection

Step 1. Based on constraint definition of closed semiring, automatically generate type collection X of type parameters: \( \{ \text{integer}, \text{real}, \text{char}, \text{boolean} \} \).

Step 2. Automatically generate operational collection Y of type parameters: \( \{ \text{MIN}, \text{MAX}, +, - , \times, /, \ast, \odot, \oplus, \ominus, \preceq \}, \preceq \}\).

Step 3. Operating parameters of Closed-semiring constraint are refined from Basebinarryop, so generate operational collection Z of Operating parameters :\( \{ \text{MIN}, \text{MAX}, +, - , /, \ast, \odot, \ominus, \preceq \}, \preceq \}\).

Step 4. Instantiation types: integer and boolean both belong to type collection X.

Step 5. Generate operational collection S of instantiation types: \( \{ \text{MIN}, \text{MAX}, +, - , /, \ast, \odot, \ominus, \preceq \} \).

Step 6. It can be concluded that operational collection \( S \subseteq \) operational collection Y.

Step 7. Operating parameters of instantiation: MIN, MAX, +, \( \ast \) and \( \odot \) belong to operational collection Z.

Step 8. As shown in step 4, 6, 7, there is no error in instantiation parameters detection.

C. Generation of C++ Programs

Step 1. Kleene generic algorithm is described in Apla language.

Step 2. Make lexical analysis of Kleene generic program, and then convert the program to a token sequence.

Step 3. Make syntax check and structure analysis of the token sequence by recursive descent parser.

Step 4. As Kleene algorithm is a generic program, start constraints matching detection. It includes formal parameters detection and instantiation parameters detection.

Step 5. As shown in section A and B, there is no error in formal parameters detection and instantiation parameters detection. Therefore, transform the right Apla Kleene program to C++ program automatically.

VI. CONCLUSION AND FUTURE WORK

The paper illustrates the current research situation of generic constraints. They mostly do support for syntax requirements, but not for semantic requirements [14], which are narrow constraints only. This paper presents a new description of generic constraints of Apla language, then gives one typical case-generic Kleene algorithm, and verifies Closed-semiring constraint. Finally uses the C++ generator to realize generic Kleene algorithm and its constraint mechanism. It has been found that the approach does not only support for proper semantic requirements, but also markedly improves dependability and safety of generic programming.

Through the design of generic constraints mechanism, we have done comprehensive tests on Apla constraints mechanism. Results show that the ability to describe and applicability to various cases have reached the design target. However, it should be point that we need develop more cases for absorbing experiences in using our approach, then to determine how widely used it can be.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Major Program, Grant No. 61020106009), Natural Science Foundation of Jiangxi Province of China (Grant No.20122BAB211030) and Science and Technology Project of Education Department in Jiangxi Province(Grant No.GJJ12199).
REFERENCES


Zhengkang Zuo, born in 1980. He is a Ph.D. candidate in computer science from the Institute of Software, Chinese Academy of Sciences, Beijing, China. He is also student membership of China Computer Federation. His current research interests include formal methods and software automation. He has published papers in a number of core journals (such as Computer Engineering & Science, Computer Engineering) and conferences (such as CARPI2012, ICCDA2011, FAW2009, FAW2008).

Jinyun Xue, Professor. He is doctoral tutor of Institute of Software, Chinese Academy of Sciences, Beijing, China and senior member of China Computer Federation (CCF). His current research interests include formal methods and software automation.

Changjing Wang, Associate professor. He received a Ph.D. in computer science from the Institute of Software, Chinese Academy of Sciences, Beijing, China. His current research interests include formal methods, software automation and software specification methods.