Egyptian Informatics Journal (2010) **11**, 1–10

Cairo University

Egyptian Informatics Journal

www.elsevier.com/locate/eij
www.sciencedirect.com

ORIGINAL ARTICLE

DisBlue+: A distributed annotation-based C# compiler

Samir E. AbdelRahman [a,*], **Amr M. AbdelLatif** [b]

[a] *Faculty of Computers and Information, Cairo University, Egypt*
[b] *Faculty of Computing and Informatics, Zagazig University, Egypt*

Received November 2009; accepted 2 May 2010
Available online 17 July 2010

KEYWORDS

Annotations;
Axiomatic semantics;
Distributed compiler;
Blue

Abstract Many programming languages utilize annotations to add useful information to the program but they still result in more tokens to be compiled and hence slower compilation time. Any current distributed compiler breaks the program into scattered disjoint pieces to speed up the compilation. However, these pieces cooperate synchronously and depend highly on each other. This causes massive overhead since messages, symbols, or codes must be roamed throughout the network. This paper presents two promising compilers named annotation-based C# (Blue+) and distributed annotation-based C# (DisBlue+). The proposed Blue+ annotation is based on axiomatic semantics to replace the if/loop constructs. As the developer tends to use many (complex) conditions and repeat them in the program, such annotations reduce the compilation scanning time and increases the whole code readability. Built on the top of Blue+, DisBlue+ presents its proposed distributed concept which is to divide each program class to its prototype and definition, as disjoint distributed pieces, such that each class definition is compiled with only its related compiled prototypes (interfaces). Such concept reduces the amount of code transferred over the network, minimizes the dependencies among the disjoint pieces, and removes any possible synchronization between them. To test their efficiencies, Blue+ and DisBlue+ were verified with large-size codes against some existing compilers namely Javac, DJavac, and CDjava.

* Corresponding author.
E-mail addresses: s.abdelrahman@fci-cu.edu.eg (S.E. AbdelRahman), amro43210@gmail.com (A.M. AbdelLatif).

Peer review under responsibility of Faculty of Computers and Information, Cairo University.
doi:10.1016/j.eij.2010.06.001

ELSEVIER **Production and hosting by Elsevier**

1. Introduction

The research discipline is to enhance compilation construction phases to speed up the compilation processes using simple concepts, namely axiomatic semantics, annotations, and distributed compiled interfaces.

Object-Oriented compiler construction phases [1] are clustered in two major groups namely, front-end and back-end stages. The front-end cluster has Lexical Analyzer to scan the code, Syntactic Parser to parse scanned code generating one main Abstract Syntax Tree (AST), and Semantics Analyzer to add all program semantics to the AST. The research

scope is only the front-end phase to help the developer in tackling this defect.

Axiomatic semantics [2] is commonly associated with proving a program/instruction to be correct using a purely static analysis of the program text. A statement correctness is described by placing assertions before and/or after the statement to present pre-conditions and post-conditions, respectively, written as '<PRE, POST> S'. Always, PRE is interpreted as a condition to execute the related statement and POST, if violated, the program is terminated.

Annotations [3–15] are added to the program in order to assert extra information at any particular code point. The developer may use his/her annotation(s) to extend the features of the programming language and/or its compiler. Such extension is used as tag(s) to make the compiler handle these features. However, adding more tags to the program result in more tokens to be parsed which leads to a compilation overhead. Till now, all innovative annotations suffer from this defect.

Distributed/parallel compilers [16–31] are appeared to speed up the compilation process. It can be done in one or more compiler phases such as scanning, parsing, semantic analysis, or also through pipelining between them. Recently, most of the research on distributed compiler [20,21,23,30] have been conducted based on modularity. Either the separation between modules or the separation between module prototype (declaration) and implementation (definition) is always used to reduce the compilation time. When distributing these pieces on remote sites, each site has to wait for specific information from the other sites in order to continue its compilation successfully. Such synchronization increases the compilation time. Moreover, network overhead occurs due to the large amount of code that is transferred among the cooperative remote sites. As per our knowledge, all current distributed compilers suffer from these defects.

To undertake all above mentioned flaws, a C# compiler, named Blue [32], was first selected as the research baseline. Second, Blue+ was implemented to handle the proposed annotation. Third, DisBlue+ was built over Blue+ to achieve all above contributions. Finally, many applications were applied to validate the proposed ideas. To test Blue+ and DisBlue+ efficiencies, Java/Blue and DJavac/CDJavac compilers [21,23] were used, respectively.

The remainder of this paper is as follows. The main research contributions and goals are divulged in the following section. Section 3 covers some related works. Section 4 illustrates by examples Blue+ aspects. Section 5 demonstrates by examples DisBlue+ features. Section 6 shows some implementation issues for the two proposed compilers. In Section 7, experimental results are presented. To end the paper, conclusions are drawn in Section 8.

2. Research contributions

Axiomatic semantics, annotations, and compiled interfaces concepts are existing current programming issues (Section 3). However, this research novelty is how to utilize these notions to acquire the research goal. Besides code readability and reusability, the research goal is to overcome the overhead because of annotation tagging and/or the distributed compilation. The proposed utilization could be summarized as follows:

(1) All program conditions could be replaced by axiomatic semantics as precompiled annotations. These annotations are created and compiled by the developer at development time before the compilation takes place. In addition, they are reusable and more readable than any current conditional (compound) statements. Hence, the total produced tokens are reduced and the updates of them are easier than ever.
(2) In the proposed distributed compilation environment, compiled interfaces are only roaming the network such that dependencies among the corresponding components are asynchronously assured with highly automatic management. Also, since the interfaces are light, the total transferred bytes are reduced compared with nowadays distributed compilers.

To gauge the research contributions, the effectiveness of the proposed utilization for compilation time speed was measured (Section 7). Additionally, its ease of use, readability, and reusability were validated using many real-life as well as simple applications (Sections 4 and 5).

3. Related work

Many research efforts in compilers [8,17,30] have been conducted introducing annotations or distributed techniques to enhance the compilation phases. In the following, some related researches are described.

3.1. Annotations

Annotations can be added in the form of metadata, which is then made available in later stages of building or executing a program. For example, a compiler may use metadata to make decisions about what warnings to issue [6]; a debugger may use assertions, mixed with source code to tackle the behavior of program module [9]; an optimizer may use them also to allocate registers [7]; a linker [11,33] can use metadata to connect multiple object files into a single executable one; a visual designer [11,33] may use metadata to control the visualization of module attributes. Modern programming languages, as well as traditional ones, support program annotation. Java and .NET enable annotation to pass through all compilation stages. Early releases of Java permitted the use of annotations in the class file named attributes [10]. Also for performance optimization purposes [7] and array bound check elimination [21], annotations on byte-code were applied. Java 1.5 introduces formally annotations on the language level [1,34]. NET formally uses annotations, as attributes, from its first release [33–37].

Recently [15] presented a classification for annotations depending on their usage. In this classification, annotations can be imperative, indicative, or subjunctive. Imperative annotations are commands executed at runtime [34]. For example, the @WebMethod annotation can be attached to a java method to convert the method to be web service method. Indicative annotations are the facts that can be stemmed and deduced from the program code [9]. Checking array boundaries is an example. Subjunctive annotations are tags having the form of pre-/post-conditions (assertions) [5,9,13] that when they are executed and the execution violates, a runtime exception is thrown. For example, "require" and "ensure" are two

keywords used as program annotations in ALL [8] and Eiffel [12] languages for pre-/post-condition, respectively.

All annotations including pre-/post-conditions, applied so far, add overhead to the compiler as more tokens are required to be parsed. This paper proposes, using axiomatic semantics (Section 1), a new type of subjunctive annotation by which no extra information added to the source code and at the same time applies conditions (checks) on program statements.

3.2. Distributed/parallel techniques

Many distributed/parallel compilers have been constructed to reduce the compilation time. A parser may decompose the grammar into small disjoint parts and make each processor responsible for one part [8,16,17,25,26,28,29]. Partitioning the parse tree into sub-trees and distributing them on a set of processors is proposed in which each processor performs the semantic analysis and code generation, then, the result is finally gathered.

Refs. [20,21,30] have been built based on the concept of modularity. Zuse [20] is a modular language that introduces the concept of concrete, abstract, semi-abstract types and permits the module to be partitioned into specification and implementation. Using Zuse compiler, the master first distributes compilation units to all slaves (clients) such that they must begin each phase at the same time. Second, each slave generates its own code. Finally, the master combines all code generated into a complete executable program. However, Zuse compiler faces several defects. First, it prevents a developer to write two correlated modules. Second, synchronization and dependency problem prohibit slaves to continue independently. The problem occurs when one slave needs some addressing information or inline methods from the other(s); that is why the compiler makes all slaves start each phase at the same time. Finally, network overhead occurs due to the massive addressing/code that may be exchanged between slaves.*prmc* compiler [30] distributes the compilation units to network sites such that each site extracts an external model of its class to describe the related interface. This external model is needed by dependent site(s) to continue its/their compilation. Each external model is compiled in the form of binary code which is shipped with its debugging information and related symbol table to the linker. Hence, prmc compiler suffers from the second and third defects of Zuse compiler.

DJavac [21] is the first distributed Java compiler. It parses and analyzes source files to extract information about relationship between files, which is used to build an adjacency matrix. For example, class A has a direct dependency on class B if A inherents from B, A implements B, or A has an object of B. Hence, A cannot be compiled until B has been compiled first. Once adjacency matrix is built, a scheduler is used to select ready files to be compiled first. A thread is created for each host that is responsible for communication with a single remote server. Each of these threads creates a TCP socket connection to the remote machine in order to transfer information and files. There are two types of transferred files with extensions .java and .class files. For a .java file, the transfer is from a workstation to a server for compilation. Then a .class file is transferred back to the workstation. The scheduler job is to minimize the compilation time by selecting individual/group of files to be compiled at one of the server machines. A ready file is the file that all its children are compiled and hence it becomes a leaf node in the matrix. Once a ready file is compiled, it will be removed from the matrix. This makes all files depending only on it be leaf nodes and then are nominated to be the next selection. The process continues until all files are compiled. The major DJavac drawback is the extensive use of file transfer. Additional overhead is added in transferring a class file(s). For example, if class A depends on class B and the scheduler decides to compile class A, so either A.java or class B is transferred since A cannot be compiled separately without the existence of B .java/.class file(s).

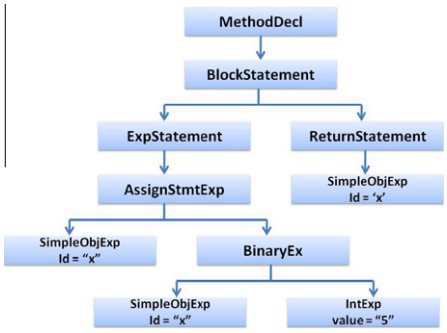

Figure 1 AST without if-statement sub-tree.

This paper proposes a distributed compilation technique that does not suffer from the synchronization problem; also it causes minimum network overhead. As the proposed compiler has some similar ideas to DJavac, it is decided mainly to compare with it.

4. The proposed annotation-based C# (Blue+)

While designing Blue+ compiler, the three main decisions are taken. First, axiomatic semantics [2] is decided to be the annotations to replace the program conditions/loops. This is because of its simplicity, readability, and declarative natures. Second, C# [33,35,36] is selected as a good OOP language. In comparison to Java and C++, Ref. [38] presented good reasons to nominate C# for this selection. Finally, as C# compiler, Blue [32] is selected to be the research baseline. It is a managed compiler that follows all OOP compiler principles [1] and was originated for educational and research purposes.

Blue+ basic idea is to utilize the pre-parsing operations in its compilation phases. That is to parse some source code before parsing time. Blue+ does pre-parsing to parse axiomatic semantics at development time. Then the generated tree is merged (injected)[1] into the parse tree of non-axiomatic statements during the parsing phase. The pre-parsing is surely designed to reduce compilation time. Example 1 illustrates this idea by showing the Abstract Syntax Tree (AST) before and after using if-statement to enclose the expression. The if-statement sub-AST is marked by surrounded thick closed line.

Example 1. Given the two non-equivalent codes below (Figs. 1 and 2) then the corresponding ASTs are depicted in Figs. 3 and 4, respectively. Some potential clues should be rendered as follows:

[1] The word injection is used in the context since the axiomatic semantics parse tree is not totally merged with non-axiomatic one but it is instilled at one of its node.

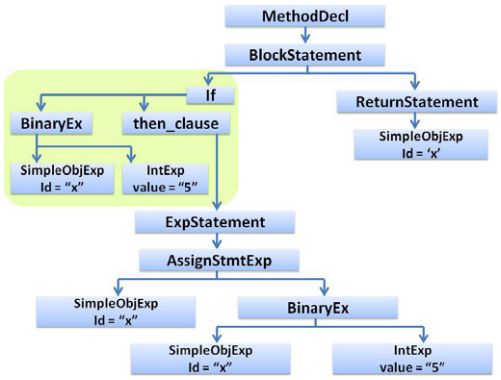

Figure 2 AST with if-statement sub-tree.

```
static int function(int x)
{  x = x + 1;  return x; }
```

Figure 3 Fig. 1 code.

(1) Fig. 4 code should be interpreted as Fig. 3 code coupled with a constraint on "x" values. Thus, in Fig. 4 code, the statement "if (x < 0)" is the assertion on "x = x + 1" statement through which the value of "x" is ensured to be negative.
(2) While Fig. 1 depicts the AST of "x = x + 1" statement, Fig. 2 portrays the AST of two statements namely "if (x < 0)" and "x = x + 1".
(3) The difference between these two figures is only the surrounded sub-tree in Fig. 2 which is related to "if (x < 0)". This means that if this tree is cut, then both figures are syntactically identical. Also, such a sub-tree may be used as an independent reusable part.
(4) If Fig. 2 surrounded sub-tree is cut and handled separately from the remaining parts, then the whole figure could be achieved again by doing two injections. The first injection is that Fig. 1 < Expstatement > node is joined to < then_clause > node of "if" sub-tree. Then, the second one is met by connecting the first injection with Fig. 1 < BlockStatement > point.

The above clues triggered the proposed Blue+ idea as follows:

(a) At program development time:

- The developer may write (separately) the assertion which (s)he intends to use instead of the if-statement. While writing it, the assertion is compiled into its corresponding AST, Fig. 2 surrounded sub-tree only, and is kept in an external file.
- After that, when the developer wishes to (re)use that assertion in his code, he restores and reloads the file in the memory. Consequently, the developer may easily reuse or modify this assertion to further usages. This results in rapid maintenance procedures.

(b) At program parsing phase, the assertion tree is placed in its related position to form one unique parse tree.

By doing the above steps, the time consumed to tokenize the conditions is removed especially if the compiler editor, as an interpreter, supports developing them visually at the same time they are tokenized and parsed.

The pre-parsing phase could not be done unless the programming language grammar is modified to support axiomatic semantics rules. Blue+ adds axiomatic semantics rules Fig. 5 to C# BNF.

Many real-life applications [39] are implemented to prove the feasibility of the Blue+ ideas. Also, they disclose how the research beliefs are very important to the developer to code/update the program easily, simply, and readably with minimum managerial efforts. Some remarks should be noticed as follows:

(1) The Blue+ GUI, which has functionalities similar to Excel function editor, simplifies the writing of annotations (assertions).
(2) The resultant Blue+ code is much simpler and smaller than its Blue synonym.
(3) The axiomatic semantics declarative and simplicity natures give the Blue+ code its readability feature especially if the developer uses many (repetitive compound) conditions/loops in his/her code.

In the following two applications are selected, namely Travelling Sales Problem and Scanner Generator for Infix Expression, to show the Blue+ feasibility, readability, and reusability features.

4.1. Travelling Sales Man Problem

Table 1 shows the Blue+ assertions and their related Blue conditions in Fig. 6. The following points should be clarified from Table 1:

```
static int function(int x)
{ if(x < 0)  x = x + 1;   return
```

Figure 4 Fig. 2 code.

```
BlockStatement → '{'
      ( decl |
        PrePostNode statement ( elseNode statement ) ? )*
  '}'
Statement →
          ObjExp '.' id '(' explist ')' ';'   // method call
  | return [ exp ] ';'
  | SwitchStatement
  | ForeachStatement
  | ForStatement
  | TryCatchFinallyStatement
  | BlockStatement        // blocks can be nested
decl →
     TypeSig id ';'          // declare a local variable
        | TypeSig id '=' exp ';'
PrePostNode →  '<' ( id ? , id ? ) '>'
```

Figure 5 Augmented Blue grammar with axiomatic semantics.

Table 1 Blue+ Fig. 7 assertions and related Blue conditions.

No.	Fig. 7 assertions	Fig. 6 equivalent code conditions
1	<isnPositive,>	if (n > 0)
2	<countn,>	for (int i = 2; i < = n; i + +)
3	<allsubsets,>	for (int subset = 1; subset < A.size()-1; subset + +)
4	<subsetNotInA,>	for (int i = 2; i < = n && ! contains(A[subset],i); i + +)
5	<else>	else

(1) Because of their declarative notions, all Fig. 7 assertions are more readable than their Fig. 6 related constructs, e.g. assertion 4 is more readable and simpler than the related for loop complex condition.
(2) When each assertion parse tree is injected, the visibility and scopes of all parse tree variables and methods, such as i, n, subset, A.size(), and contains (A[subset],i), are checked with all main AST elements.
(3) Separately from the other code parts, all assertions are coded and parsed first in the development time. Thereafter, when the developer needs to use/repeat any condition before or after his code statement, he may choose the assertion that encodes the required condition. Assumingly, if the developer wishes to repeat Fig. 6 code fourth condition statement many times, e.g. 1000 times, then he must write 1000 statements of it. This leads to parse the loop 1000 times which increases the Lexical Analyzer and Syntactic Parser time. Also, if it is required to change some part of it, then the developer must do this adaptation 1000 times. Compared with this approach, if Fig. 7 assertion is used, then the statement is parsed and modified once. That leads to decrease the time of both phases and consequently the maintenance procedures are reduced.
(4) In the parsing time, the Blue+ parser matches <else> with the nearest 'condition' axiomatic semantic type. If the parser could not make such match, then it generates parser error. In the given example, <else> assertion is matched to <isnPositive,>.
(5) If the developer wishes to rename any assertion, then he updates it once in the file of assertions and Blue+ itself renames all affected assertions in the whole related programs.

```
void travel (int n, int [][]W,  ref int minlength)
{ if ( n > 0) {
   int [] V = getAllVertices();
   int [][] A = getSubsetCombination();
   int [,]D = new int[n,A.size()];
   for (int i = 2; i <= n; i++)     D [i,0] = W[i,1];
   for (int subset = 1; subset < A.size()-1; subset++)
   for (int i=2;i <= n  && ! contains(A[subset],i) ; i++)
   {  D [i][subset] = minimum (W, D,A,i,subset); }
   D [1][subset] = minimum (W,D,A,2,subset);
   minlength = D[1][subset]; }
   else   return; }
```

Figure 6 Blue Travelling Sales Man Problem code.

```
void travel (int n, int [][]W, ref int minlength)
{
  <isnPositive,>  {
   int [] V = getAllVertices();  int [][] A = getSubsetCombination();
   int [,]D = new int[n,A.size()];
   <countn,>   D [i,0] = W[i,1];   <allsubsets,>
   <subsetNotInA,>   { D [i][subset] = minimum (W, D,A,i,subset); }
   D [1][subset] = minimum (W,D,A,2,subset);
   minlength = D[1][subset];
 } <else>    return;}
```

Figure 7 Blue+ Travelling Sales Man Problem code.

```
infixExpr → expr+
expr   → term ADDOP expr
term   → factor MULTOP term
factor → LPAREN expr RPAREN | NUMBER
NUMBER → ([0-9])+ ('.' ([0-9])+)?
ADDOP  → - | +
MULOP  → * | /
LPAREN → (
RPAREN → )
EOL    → \n
```

Figure 8 Context free grammar for Infix Expression.

```
 public Token GetNextToken(){
int ch;
Token t = new Token(); state = 1;
while(true){
  switch(state){
          case 1:
            ch = (char)stream.Peek();
            t.val += (char)ch;
            if(ch >= '0' && ch <= '9') state  =  2;
            else if(ch == '+' || ch == '-') state = 6;
            else if(ch == '*' || ch == '/') state = 7;
            else if(ch == ' ' || ch == '\t') state = 8;
            else if(ch == '\n') state = 10;
            else if(ch == 65535) state = 11; //no more tokens
            else fail();  break;
...  } } }
```

Figure 9 Slice of DFA code by Blue.

```
public Token GetNextToken(){
int ch;
Token t = new Token();
state = 1;
<while_true, > {
  switch(state){
    case 1:
      ch = (char)stream.Peek();
      t.val += (char)ch;
     <DIGIT,> state  =  2;
     <else> <ADDOP,> state = 6;
     <else> <MULOP,> state = 7;
     <else> <WS,>   state = 8;
     <else> <EOL,>   state = 10;
     <else> <EOF,>   state = 11; //no more tokens
     break;
    … } } }
```

Figure 10 Slice of DFA program by Blue+.

4.2. Scanner Generator for Infix Expression

Scanner is part of a compiler and is typically written in a certain programming language. For purpose of demonstration a small grammar is selected for illustration Fig. 8. In Fig. 8, the above grammar, such as NUMBER, LPAREN, RPAREN, EOL, ADDOP, and MULOP, are shown. The related DFA codes are listed in Figs. 9 and 10.

5. The proposed distributed annotation-based C# (DisBlue+)

Built on the top of Blue+, DisBlue+ inherits all its features (Section 4). Moreover, DisBlue+ has own distributed features that totally depend on compiled interface concept. The DisBlue+ idea is based on OOP features coupled with software engineering principles. With support of polymorphism and interface features, the application software engineer forms the application as a set of correlated interfaces in the application server. The compiled interface (CI) coupled with its compiled relevant interface(s) is sent to one of the application client. In that client, the compiled interfaces are extracted to get their specifications via reverse engineering module and then the body of CI is written. These steps are rendered as follows:

(1) The software engineer divides the program into a set of tasks and then specifies the skeleton code for each task. The skeleton code is defined in terms of a set of interfaces.
(2) The software engineer specifies which task will be allocated to a specific developer.
(3) Each task visibility is determined by the software engineer.
(4) A distribution command is issued. A compilation occurs at the server for each task and sent to each developer (Client).
(5) Upon task reception by each client, it is the responsibility of each developer to make the implementation class by implementing the received interface.
(6) When the developer finishes writing his own task, he compiles the task with Blue+ compiler into .dll file and sends it to the server.
(7) The server collects all task implementations and compiles them all into a complete program.

In spite of the fact that the compiled interface is not a novel idea, the manner of its utilization in the above steps is a distributed compiler novel idea and it reveals many potential points. First, the divided tasks could be asynchronously coded without any synchronous collaboration among the project developers. Second, no conflict or possible programming miscommunications may be arisen because all interfaces and related bodies are well defined by the software engineer. Third, all tasks are automatically assembled which gives better software management activities than any today distributed compilers. Finally, since most DisBlue+ transferred files are compiled interfaces, the total transferred bytes of DisBlue+ application is small. That is in contrast to similar nowadays compilers which each of them makes its clients exchange the whole (compiled) class(s) among each other because of their dependencies.

In the following two applications are selected, namely Compiler Generator and Expert System, to show how the dependencies among cooperative classes could be easily and readably managed by DisBlue+ mentioned features.

The compilation phases can be developed as separate compilation units such that each phase could be presented by one object which calls the objects' methods. So if each phase interface is clearly defined, the dependent phases can be developed separately without the need of that phase definition. With well defined scanner and parser interfaces, the dependency between

```
public class Token
{ public string val; public ITokens tokenType;}

public enum ITokens
{ NUMBER = 1, ADDOP = 2, MULOP = 3, EOL = 4, LPAREN = 5,
RPAREN = 6, WS = 7, EOF = 8}

public interface IScanner
{Token GetNextToken();}

public interface IParser
{ void Begin(IScanner scanner);}
```

Figure 11 DisBlue+ compiler interfaces and tokens.

```
class Parser:IParser
{
  private Token lookahead;
  private IScanner scanner;
  public void Begin(IScanner scanner)
  { this.scanner = scanner;
    lookahead = scanner.GetNextToken();   Infix();
  }
  private void Infix()
  { {
      Expr();  eat(ITokens.EOL);
      Console.WriteLine();
      // test the lookahead is left paranthesis or number
      <lookaheadTypeLPAREN_OR_NUMBER,>
        break;
    } <,while_true>
    eat(ITokens.EOF);
    Application.Exit();
  } ..}
```

Figure 12 The slice of DisBlue+ parser implementation.

them can be partially removed during the development. Fig. 11 present Tokens, Types of Tokens, Scanner, and Parser specifications as Token class, ITokens enum, ISanner interface, and IParser interface, respectively, such that the Token and IToken are the shared data structures. After the coordinator distributes all the necessary compiled interfaces, the two concurrent developers may code the implementation for them independently. The scanner and parser implementations are shown in Fig. 12.

An Expert System (Fig. 13) may be implemented as three components mainly Knowledge Rules, Knowledge Memory, and Knowledge GUI. These components may be distributed to Knowledge Rules Class, Knowledge Memory Interface, and Knowledge GUI class, respectively, such that the interface is shared between two classes. It is the responsibility of the software engineer to define these components' specifications.

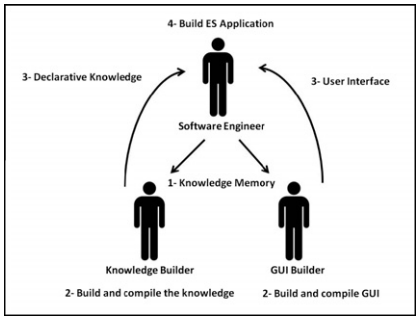

Figure 13 Expert System application DisBlue+ design.

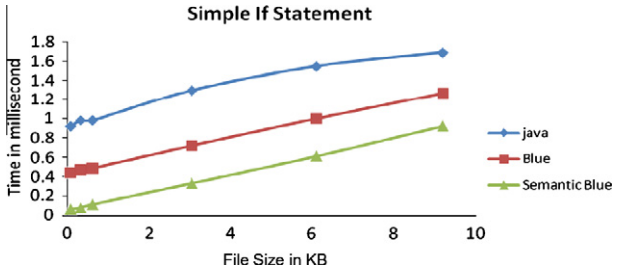

Figure 14 Comparison of Javac, Blue, and Blue+ compilers' time.

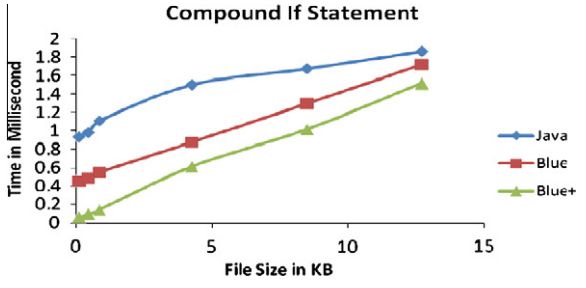

Figure 15 Comparison among Javac, Blue, and Blue+ compilers' time.

6. Implementation highlights

Blue is an open source C# compiler that uses the namespace **System.Reflection.Emit()** which is used to generate MSIL (MicroSoft Intermediate Language). Blue does not use the **System.CodeDom** namespace to parse the code; instead it builds its own lexer, parser, symbol table, and code generator. The parser of Blue was modified to cope with the new syntax of pre-/post-conditions. Blue was embedded with the research IDE to avoid the cost due to startup time for .NET framework. An IDE is used to visually build the axioms (pre-/parsed-conditions). The developer can add axioms at any time before, during or after editing the program. The AST axioms are serialized and saved in an external file.

DisBlue+ is the distributed compiler that enables many developers to share their work with some coordination from a software engineer who works from a server. The server allocates a thread for each client that connects to it. .NET Remoting is used to achieve the communication between objects. **RObject** is an object that is published by the server; this object contains many networking methods for connecting, disconnecting, sending and/or receiving files from/to clients. Any client wants to participate on the project can request a connection. The server accepts the connection. Each client can work while he is offline and sometime later he can submit his task.

7. Experimental results

To test Blue+ and DisBlue+, two code generators were developed and used to generate the code examples for both Blue+

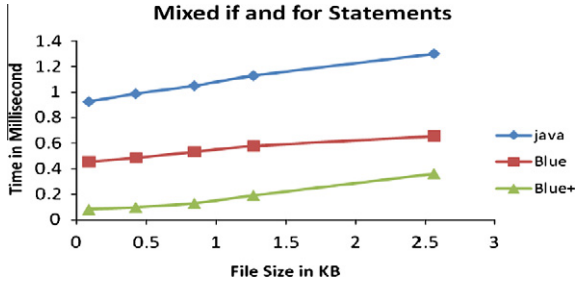

Figure 16 Comparison of Javac, Blue, and Blue+ compilers' time.

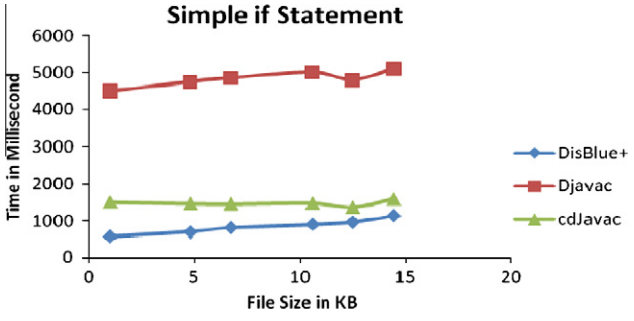

Figure 17 Comparison of DisBlue+, DJavac, and CDJavac compilers' time.

Table 2 Blue+ compiler with simple if condition.

Number of pre-conditions	File size (bytes)	Java time (ms)	Blue time (ms)	Blue+ time (ms)
Simple if-statement				
10	683	0.921875	0.4375	0.0625
50	3174.4	0.984375	0.46875	0.078125
100	6082.56	0.984375	0.484375	0.109375
500	30412.8	1.296875	0.71875	0.328125
1000	60928	1.546875	1	0.609375
1500	91955.2	1.6875	1.265625	0.921875

Table 3 Blue+ compiler with compound if condition.

Number of pre-conditions	File size (bytes)	Java time (ms)	Blue time (ms)	Blue+ time (ms)
Compound if-statement (five conditions)				
10	924	0.9375	0.453125	0.0625
50	4280.32	0.984375	0.484375	0.09375
100	8478.72	1.109375	0.546875	0.140625
500	42393.6	1.5	0.875	0.609375
1000	84889.6	1.671875	1.296875	1.015625
1500	126,976	1.859375	1.71875	1.515625

Table 4 Blue+ compiler with mixed for and if conditions.

Number of pre-conditions	File size (bytes)	Java time (ms)	Blue time (ms)	Blue+ time (ms)
Mixed if- and for-statements				
10	892	0.921875	0.453125	0.078125
50	4229.12	0.984375	0.484375	0.09375
100	8417.28	1.046875	0.53125	0.125
150	12697.6	1.125	0.578125	0.1875
300	25,600	1.296875	0.65625	0.359375

Table 5 DisBlue+ compiler vs DJavac and CDJavac.

Cases	Total project size (kbytes)	DisBlue+ total time (ms)	Random1 benchmark	
			DJavac total time (s)	CDJavac total time (s)
1	9850.88	0.575	4.5	1.50
2	47718.4	0.700	4.75	1.46
3	66969.6	0.811	4.85	1.45
4	105,472	0.900	5.009	1.47
5	124,928	0.966	4.8	1.36
6	144,384	1.139	5.1	1.59

and DisBlue+, respectively. The code was randomly generated and hence axioms with different types might be (not) nested and/or (not) reused.

In Blue+ experiments, three case studies were used namely: 'simple if' (Table 2 and Fig. 14), 'compound if' (Table 3 and Fig. 15), and 'mixed if and for' (Table 4 and Fig. 16). Each case was measured based on the number of axiomatic semantics (pre-conditions) that were coded. The total program size in bytes varied from 600 to 126,976; 145,000 bytes is the maximum file size that may be compiled by Blue. Time was measured in ms and comparisons were made against Javac and Blue compilers. Two main points should be noticed. First, the time was measured at the worst case in which the application had no reused axiom; no axiom was repeated. Second, since Blue+ always replaces the compound condition with a single reference, the condition length is irrelevant.

In DisBlue+ experiments (Example 2), DJavac and CDjava algorithms [21,23] were implemented and they were validated with [21,23] experiments. Thereafter, their benchmark, named 'Random' [21,23], was selected to make the comparison among the three compilers. The benchmark was selected because it got the fastest DJavac and CDjava results [21,23]. The research results were calculated on programs having sizes in bytes varying from 9850 to 144,384. In each compiler, its time was estimated purely based on Eq. (1) and was measured in ms.

All experiments were done on a machine with processor Pentium 4 CPU 2.40 GHz and 256 of RAM for Blue+. While, DisBlue+ tests were done on a server and four clients with the

same specifications having different operating systems of Windows XP, Linux Fedora Core 6, and Windows Server 2003. Each distributed compiler run was on two Hyper-threaded multi-processor machines of 667 MHz and 3 GHz speeds coupled with 2G RAM for each.

Example 2 (Table 5 and Fig. 17). *DisBlue+* time depends on the file transfer and the compilation at network sides (Eq. (1)).

$$\begin{aligned} Total\_Compilation\_time = {} & Serer\_task\_compile\_time \\ & + Serer\_task\_distribution\_time \\ & + Max(client\_compilation\_time) \\ & + Max(client\_task\_send\_back) \end{aligned} \tag{1}$$

All tasks are developed and compiled simultaneously. Also they are compiled independently via OOP interface construct. Hence, the total compilation time is dependent on the maximum of all clients. Furthermore applying axiomatic semantics on each client would also reduce compilation time.

8. Conclusions and future work

Blue+ and DisBlue+ presented ideas that accelerated (distributed) OOP compilation and maintenance phases. These notions are very important on account of the following reasons:

- The program code becomes more readable and reusable using axiomatic semantics instead of program (rhythmic compound) conditions.
- The highly automatic managerial efforts of distributed compilation activities are achieved using minimal programming dependency.

However, the DisBlue+ software engineer should consider the removal of all application dependencies via interfaces. It is intended to build a new C#/C++ compiler having the above features such that they may affect all compilation phases especially parsing and semantic phases. More types of annotations should be investigated to enhance such compiler.

References

[1] Appel Andrew W, Palsberg J. Modern compiler implementation in Java. Cambridge University Press; 2002.

[2] Sebesta RW. Concepts of programming languages. 8th ed. Addison-Wesley; 2007.

[3] Bloch J. JSR 175: a metadata facility for the Java programming language. Available from: http://jcp.org/en/jsr/detail?id=175.

[4] Biberstein M, Sreedhar VC, Mendelson B, Citron D, Giammria A. Instrumenting annotated programs. In: First ACM/USENIX international conference on virtual execution environments (VEE'05), 2005.

[5] Dana N. Extended static checking for Haskell. In: Haskell workshop, 2006.

[6] Flanagan C, Leino K, Lillibridge M, Nelson G, Saxe J, Stata R. Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation, 2002.

[7] Jones J, Kamin SN. Annotating Java class files with virtual registers for performance. Concurr Pract Exp 2000;12(6):423–44.

[8] Khurshid S, Marinov D, Jackson D. An analyzable annotation language. In: OOPSLA 2002, vol. 25, 2002. p. 231–45.

[9] Leavens G, Baker A, Ruby C. Preliminary design of JML: a behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. [Last Revision: Aug 2001].

[10] Lindholm T, Yellin F. The Java virtual machine specification. 2nd ed. Addison-Wesley; 1999.

[11] Meredith P, Pankaj B, Sahoo S, Lattner C, Adve V. How successful is data structure analysis in isolating and analyzing linked data structures? Technical Report #UIUCDCS-R-2005-2658, Computer Science Department, University of Illinois, December 2005.

[12] Meyer B. Eiffel: the language. New York, NY: Prentice Hall; 1992.

[13] Newkirk J, Vorontsov A. Design: how .NET's custom attributes affect design. IEEE Softw 2002;19(5):18–20.

[14] Pechtchanski I, Sarkar V. Immutability specification and its applications. In: The joint ACM Java grande – ISCOPE 2002 conference, November 2002. ACM Press; 2002. p. 202–11.

[15] Würthinger T, Wimmer C, Mössenböck H. Array bounds check elimination for the Java HotSpotTM client compiler. In: Proceedings of the fifth international symposium on principles and practice of programming in Java. PPPJ; 2007. p. 125–33.

[16] Asthagiri CR, Potter J. Associative parallel lexing. In: Proceedings of the sixth international parallel processing symposium, Beverly Hills, CA, USA, March 1992.

[17] Baccelli F, Fleury T. On parsing arithmetic expressions in a multiprocessing environment. Acta Inform 1982;17.

[18] Baccelli F, Mussi P. An asynchronous parallel interpreter for arithmetic expressions and its evaluation. IEEE Trans Comput 1986;35(3).

[19] Boehm H, Zwaenepoel W. Parallel attribute grammar evaluation. In: Proceedings of the seventh incremental conference on distributed computing systems, IEEE, September 1987. p. 347–54.

[20] Collberg C. Distributed high-level module binding for flexible encapsulation and fast inter-modular optimization. In: Proceedings of the international conference on programming languages and system architectures, Zurich, Switzerland, 1994. p. 282–97.

[21] Dalton A. Distributed Java compiler: implementation and evaluation. Master's Thesis, Appalachian State University, 2004.

[22] Das R, Wu J, Saltz J, Berryman H, Hiranandani S. Distributed memory compiler design for sparse problems. IEEE Trans Comput 1995;44(6):737–53.

[23] Distcc: a fast, free distributed C/C++ compiler; 2004. Available from: http://distcc.samba.org/.

[24] Ensink B, Adve V. Coordinating adaptations in distributed systems. In: 24th international conference on distributed computing systems (ICDCS 2004), Tokyo, Japan, March 2004.

[25] Hill J. Parallel lexical analysis and parsing on the AMT distributed array processor. Parallel Comput 1992;18.

[26] Khanna S, Ghafoor A. A data partitioning technique for parallel compilation. In: Proceedings of the workshop on parallel compilation, Kingston, Ontario, Canada, May 1990.

[27] Languages and compilers for parallel computing. In: 14th international workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1–3, 2001. Revised papers, Lect Notes Comput Sci, vol. 2624/2003. Berlin/Heidelberg: Springer; 2004.

[28] Levke K. Pila: lexical and syntactical analysis on a pipelined processor. In: Proceedings of the workshop on parallel compilation, Kingston, Ontario, Canada, May 1990.

[29] McDonald L, Wendelborn A. Parallel lexical analysis. In: Proceedings of the workshop on parallel compilation, Kingston, Ontario, Canada, May 1990.

[30] Private J, Ducournau R. Link-time static analysis for efficient separate compilation of object-oriented languages. In: The

sixth ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering (PASTE 2005), 2006. p. 20–7.
[31] Yang W. A finest partitioning algorithm for attribute grammars. In: Second workshop on attribute grammars and their applications WAGA99, March 1999.
[32] Blue: a C# compiler. Available from: http://blogs.msdn.com/jmstall/archive/2005/02/06/368192.aspx.
[33] Mono Project. Available from: http://www.mono-poject.com/Main_Page.
[34] Sun JDK. Available from: http://developers.sun.com/downloads/.
[35] ECMA C# and common language infrastructure standards. Available from: http://msdn2.microsoft.com/en-us/netframework/Aa569283.aspx.
[36] Microsoft C# language specifications, by Microsoft Corporation, 2001.
[37] Rammer I, Szpuszta M. Advanced .NET remoting. 2nd ed., Microsoft Cooperation; 2005.
[38] Performance comparison Java/.NET runtime; October 2004. Available from: http://www.shudo.net/jit/perf/.
[39] Neapolitan R. Foundations of algorithms using C++ pseudocode. third ed. Jones and Bartlett Publishers; 2004.