Joint NCC & IIS Bull., Comp. Science, 15 (2001), 65–81

An extensible analyzer of subroutines in imperative languages*

I. V. Dubranovsky

A lot of work has been dedicated to the analysis of sequential imperative programs. However, existing tools of analysis seem to lack for clarity and extensibility. That is to say, although some of them perform powerful context-sensitive dataflow analysis, their efforts are chiefly directed to the analysis of a particular programming language. In this paper, we propose a new analyzer for C++ and Java that can be easily extended to perform intraprocedural analysis of any imperative program. We discuss the ideas motivating the choice of one or another approach to analyzer implementation.

1. Introduction

It is well known that program understanding requires different information that can be obtained from the source code by using special facilities performing source code analysis. The deeper is the analysis performed, the more detailed is data of the program obtained. Actually, the efficiency of the obtained information heavily depends on the way it is presented to the user. There may be an analyzer that performs a powerful static analysis but does not have a worthwhile user interface to explore the knowledge obtained. It is obvious that weak presentation of information in such a tool reduces the advantage of deep analysis.

The discussed issue is particularly important with regard to sequential imperative programs. No one would argue that the function or subroutine is the basic element in the imperative language. This fact explains the need for studying subroutines because they finally form the whole essence of the program. Understanding of functionality and interaction of subroutines sufficiently provides understanding of the program structure and operation. Knowing the set of arguments and results of a subroutine resolves the problem of interprocedural interaction within a program. Besides, it enables the use of the stream criteria [9] to estimate the complexity of program decomposition. Knowing the order of the use of local variables helps us to trace the effect of the function call and the side effects. Let us denote this information as the flow information (FI) of a subroutine.

*Partially supported by the Russian Foundation for Basic Research under Grant 00-01-00820.

A lot of work has been dedicated to program analysis. The algorithms of analysis differ in complexity, kind, and precision of the information extracted. For instance, the analyzer FLAVERS [8] implements the dataflow analysis to facilitate program verification. The static analyzer of semantic properties of programs [4] obtains the properties in the form of term equalities by using the abstract interpretation of the program. The analyzer Wasp [7] performs a powerful context-sensitive dataflow analysis by approximation of definite variable definitions. It has been developed to analyse programs written in the mixture of the Oberon-2 and Modula-2 languages, as well as Java programs. Therefore, it seems instructive to underline the following. The algorithm designed for Wasp is undoubtedly of great value. However, reusing it for another language requires spending an appreciable amount of time and, actually, rebuilding the analyzer, since no special means have been provided for this purpose. Besides, the way the information is displayed to the user (in fact, it is dumped into a set of text files) is not demonstrative enough. We can mention in addition the visualizer HyperCode [2] developed basically for the purpose of reverse engineering. It implements the paradigm of "transparent film" and enables displaying a wide range of program relationships and attributes for any imperative language.

This observation leads us to the idea that some kind of processor deriving FI from a program without significant time and resource consumption would be attractive. An obvious requirement to the processor is that it should display the obtained information in a convenient form. In addition, it should be an easy-to-use compact tool. We will call this processor *Sap* in the sequel.

It is the objective of this paper to discuss comprehensively the proposed instrument and its functionality, thereby providing the description of how to facilitate the process of studying and understanding an imperative program. This includes outlining of the method of utilization of existing facilities. Such a method will remove the weak points of the existing facilities related to their user interface or to the absence of the latter.

A few notes on the paper organization. In Section 2, the basic ideas and the architecture of the proposed processor are reviewed. This section also introduces requirements to the processor and the application programming interface. The analyzers for the C++ and Java programming languages are described in Sections 3 and 4, respectively. The discussion concludes in Section 5, where we examine possible ways of extension of the proposed processor and provide contact information.

2. Sap processor

In this section, we discuss the *Sap* processor that is a combination of a user interface with static analyzers. We concentrate our attention on the description of ideas motivating the choice of one or another architecture principle. These principles will hereafter help us to outline the organization of the processor components from the standpoint of internal structure and user interface. Notation and conventions will also be provided to exclude any misunderstanding.

2.1. Overview: basic ideas and their effects

Let us recall that we consider imperative languages the set of which is large enough to raise problems in implementing a processor supporting the entire set. A natural solution is to support a few languages rather than the entire set, and to enable some kind of processor extensibility for understanding multiple languages. When using such an approach, we have to distinguish the terms common to all imperative languages. As it is indicated in Section 1, the subroutine in the imperative language is a fundamental object that has attributes and an internal structure. It is clear then that the processor should present subroutines to the user as objects and directly display their attributes and structure. This provides clarity and simple observation of information links. The complexity and strictness of static analysis of different languages depend in particular on the information obtained and on the kind of parser used. On the one hand, a simple parser can be used to build an internal representation adapted to a particular static analysis. On the other hand, an existing analyzer can be used to avoid implementing a custom parser and analyzer. The latter note can be taken into account when designing the processor.

We can now trace the effects of the above ideas, thereby elaborating partial requirements to the processor. First of all, the initial set of languages to be supported has to be chosen. Let us choose C++ and Java — the most popular languages at present. Next, the way of providing extensibility should be fully understood. A popular approach to solving this problem is to implement a user interface (UI) environment and a couple of initial libraries for C++ and Java. The UI environment should provide a mechanism of registration and dynamic linkage of language libraries without having to recompile the environment. The libraries should expose the same interface for subroutine analysis. For some languages, the processor may obtain FI just by studying the procedure body only and using flow-insensitive analysis. For the rest of the languages, it may perform a deeper (but still inexpensive)

analysis together with the analysis of the functions called from the analyzed subroutine.

We will perform the flow-insensitive analysis of C++ and the flow-sensitive analysis of Java to demonstrate different approaches to program analysis and implementation of language libraries. In this case, both kinds of analysis will be intraprocedural. For the same reason, none of the existing tools of automated parser construction will be used to implement the C++ library. We will try to reuse the existing Java-frontend and the static analyzer (Java Static Analyzer Wasp [7], in particular) to implement the Java library.

2.2. Notation and conventions

The notions of *defining* and *using occurrences* will be used in the sequel. They have different meanings depending on the context they are used in. In the context of syntactic and flow-insensitive analysis, the *defining occurrence* of a variable means its definition in the source program. The *using occurrence* means any other occurrence that differs from the defining one. In the context of semantic and flow-sensitive analysis, it is convenient to call the *defining occurrence* of a variable the place where it receives its value. It is also convenient to call the *using occurrence* of a variable the place where its value is read under a certain execution of the program.

In this paper, we assume the analysis to be intraprocedural. Under this restriction, the notions of argument and result are defined as follows. When speaking about flow-insensitive analysis, we consider the argument as a variable of the subroutine that has a using occurrence in its body. The result is considered as an argument that may be modified by the subroutine. The semantics in the context of flow-sensitive analysis is similar to the generally accepted one (see, for instance, [5]): a variable is called the argument if the subroutine, under a certain execution, reads its value. Similarly, a variable is called the result if the subroutine, under a certain execution, sets its value (not necessarily to each component).

It is necessary to emphasize, just to avoid any confusion, that the processor is intended to function under Microsoft Windows 98/2000 and hence it is written by using Win32 API. For this reason, processor execution is described in this paper in terms of the operating system and Win32 API routines.

2.3. Outline of the architectural elements

In Section 2.1, a suitable approach to the implementation of extensibility has been presented. The approach suggests the creation of a UI environment

and language libraries so that each library corresponds to the supported language. Let us discuss the purpose of these components. One of the main objectives of the UI environment is to display FI and allow the user to navigate easily through it. We consider that the ability to analyse the source code should not be included in the list of aims of the UI environment because it is only intended to show visually the outcome of analysis carried out by other components. Furthermore, the UI environment should be able to start analyzers residing in their language libraries, since the UI environment cannot analyse subroutines by itself.

The language libraries should provide the implementations of required algorithms, where the main algorithm is the extraction of FI from a given subroutine. It is also natural that different libraries should expose the same API (Application Programming Interface) and the same format of data exchange. This allows the UI environment not to be concerned with library contents. The UI can merely call the interface methods which have similar sets of parameters. Let us call such an interface **Sap API**. It is described in detail below.

2.4. Application programming interface

As it is noted in Section 2.3, it is necessary to unify the interaction of the UI environment component and the language libraries. To address the issue, one should specify an interface made up of methods and data types used for the exchange of information between the caller and the library. In windows programming, such an interface is usually called an *application programming interface*, or simply API. It enables writing new applications that call library routines through this interface (see, for explanation, Win32 API in [10]). The primary goal now is to determine what methods and data types should form Sap API and what kind of work those methods should carry out.

2.4.1. Interface functions

The simplest way to determine the set of methods of Sap API is to look through the functionality of the processor in order to obtain its baselines. It is obvious that one of such baselines is subroutine analysis, which should be encapsulated in a method of API. Another basic functionality, as practice has shown, may be a search for all the subroutines in the module of the program, accommodating them into a list or a tree, depending on the particular language syntax. This, for example, allows displaying the subroutines of a module in a window immediately after opening the module in the UI environment. The other API methods do not deserve our attention and are left out of the scope of the paper.

There are some special requirements to the interface methods. They should not contain complicated algorithms in the sense that their execution will require a large amount of time resources because Sap must not force the user to wait for the algorithm finishing its execution. Otherwise, the user would have to waste time waiting until the subroutine is analysed. Instead, the interface methods may create new threads of execution, start complicated algorithms on them, and immediately return. Another important remark is that the algorithms mentioned above have no rights to use global and static variables for writing in order to use the scheme of separate memory. That is to say, the scheme of separate memory helps us to avoid thread synchronization.

As a result, the pattern of an interface method has the following form. The interface method creates a new thread executing an appropriate algorithm. When the new thread finishes execution, the originator thread is notified with a message. Such a notification should be interpreted as the termination of analysis. The message provides a pointer to data that need to be displayed. The presented pattern provides the user with a possibility to analyse several subroutines simultaneously. Note that Sap can respond to user commands while analyzing subroutines.

2.4.2. Data types

The description of Sap API would not be complete without illustration of data types used to transmit the extracted information from the language libraries to the UI environment via the interface methods. On the other hand, it seems justified not to deepen in details while describing the data types in this paper. Therefore, let us at first simply present two basic types, each of which corresponds to the appropriate interface method. Next, the construction of more complicated data types becomes possible.

Note that natural considerations should be taken into account in order to generate the data types mentioned above. According to Section 2.4.1, we have got two functionalities. We should elaborate a data type for each of them. Search for all the subroutines in the module of a program, accommodating them into a list or a tree, requires that a tree node should be specified. Such a tree node may contain the following information:

- subroutine name;
- coordinates of the subroutine in the module;
- pointer to the data identifying the subroutine in the module (for example, it may be either the position of the subroutine in the module or the prototype of the subroutine);
- size of the data identifying the subroutine.

The upper two items of the list may be used (and they are actually used) to display visually the subroutine header in the UI environment. The place where the subroutine resides in the source code of the program can also be displayed. The remaining items are reserved for implementation purposes and serve as a subroutine identifier helping to locate the subroutine when the actual analysis is started. The same ideas can be applied to the data types that accompany the other functionality. We will not pay attention to them here. Instead, it is worth giving a definition of a local variable tree used to display the hierarchy of local variables of the subroutine in the UI environment. The local variable tree presents local variables for browsing in a convenient form (see Figure 1). This tree contains local variables of the subroutine and the names of statements (such as **if**, **switch**, **for**, **while**, etc.) that cause new scopes of visibility of the variables. Each vertex of the tree corresponds to a statement in the subroutine. The list of local variables declared inside the statement is associated with the appropriate vertex. Such a tree can be defined recursively.

1. Consider a statement that does not have nested statements; let it contain the declarations of local variables $v_1, \ldots, v_m$. Such a statement is a leaf of the tree. An ordered list of variables $v_1, \ldots, v_m$ is associated with this leaf.
2. If S is a statement and $v_{01}, \ldots, v_{0m_0}, S_1, v_{11}, \ldots, v_{1m_1}, S_2, \ldots, S_k, v_{k1}, \ldots, v_{km_k}$ is a sequence of declarations of variables v_{ij} $(0 \le i \le k,\ 1 \le j \le m_i)$ and statements S_i $(0 \le i \le k)$ nested in S, then the vertices $S_1, \ldots, S_k$ are descendants of S and the ordered list $v_{01}, \ldots, v_{0m_0}, S_1, v_{11}, \ldots, v_{1m_1}, S_2, \ldots, S_k, v_{k1}, \ldots, v_{km_k}$ is associated with S.
3. The body of the subroutine is the root of the tree.

3. C++ analyzer library

The next question is implementation of the analysis for the C++ language (we remind that C++ is considered in this section).

First of all, we are restricting the input language and the input program. There is a variety of different extensions of ANSI C++. Companies developing the C++ programming environment sometimes add new keywords to the C++ character set. Thus, to avoid confusion, we assume the input string to belong to the common subset of different versions of the language. The common subset is called ANSI C++[1]. This is not a rigid restriction

[1]The specification of this standard can be found in [3]. At present, this specification is close to the current one although is not ultimate.

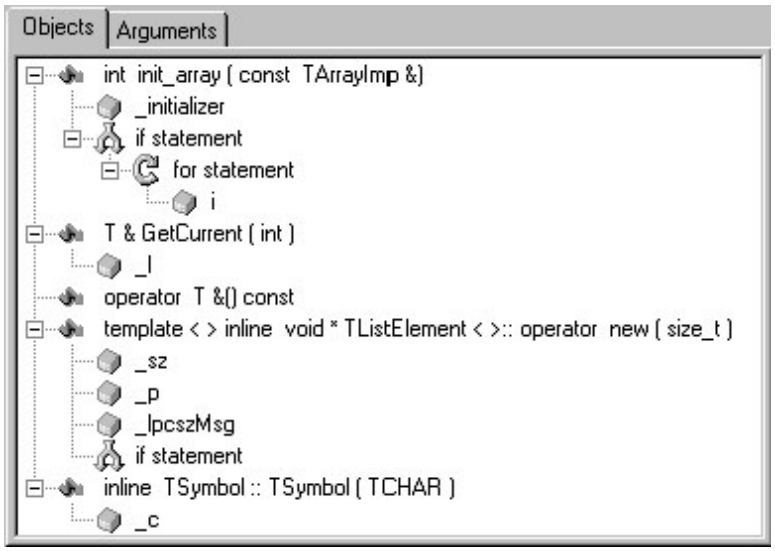

Figure 1. A window displaying the result of analysis in the form of a tree of local variables and statements

because the inclusion of new keywords is basically related to separate compilation and to platform for which programs are written. In particular, it is related to linkage specification, procedure calling conventions, etc. (e.g., the **__declspec**, **__fastcall** keywords are new with respect to the ANSI C++ standard). That is why the majority of new keywords take place outside the function definition or in function headers, which does not influence the functionality of the processor.

Another remarkable detail is that, before analyzing a program with Sap, the C++ preprocessor should process the program to perform macro expansion and conditional compilation. The reason is that we analyse solely function definitions, not considering the whole program. The macro definitions can reside outside function bodies. Therefore, the recognition of constructions hidden under those macro definitions becomes impossible. If preprocessing has not been performed, Sap provides inaccurate information.

Let us now discuss the analysis algorithm, i. e., its input, transformation, and output. The input is a string of ASCII characters that represents a phrase in C++. The output fits the description of the application programming interface presented in Section 2.4. Before we start the discussion of the transformation, we need to mention inaccuracy of FI obtained by the analyzer.

As it has been noted, the global semantic analysis is not performed. As a result, certain constructs are interpreted incorrectly (see Section 3.2), since type and function declarations are usually placed outside function bodies. Therefore, the algorithm is not aware of the majority of the declarations. To simplify the analysis, it was decided to ignore completely the type definitions, since such a restriction practically does not influence the result of the analysis. The differences between type names, function names, and other names are revealed using the right context of the name. Nevertheless, such a context does not always provide a successful differentiation. This leads to inaccuracy of flow information, that is, the analyzer treats certain types as global arguments, certain expressions as declarations, etc. (see Section 3.2). We should say that the analyzer is implemented so that the inaccuracy of obtained FI is minimized as much as possible, although it cannot be fully avoided in static analyzers.

The transformation of an input string builds a simplified derivation tree. The *lookahead recursive descent*[2] algorithm generates this tree implicitly. We should underline that a simplified tree is built, because the specificity of the problem allows us to perform incomplete parsing of some constructs. For instance, it is not obligatory to perform full parsing of **using::***unqualified_identifier*; . It is enough to skip tokens up to the semicolon.

Let us now consider the transformation at a higher level of details. It is carried out by:

- lexical scan,
- parsing,
- primitive semantic analysis.

These algorithms interact through the standard single-pass scheme of translation: the parser is the primary algorithm using the scanner to get a new token from the input string. Syntactic and semantic analyses are performed simultaneously. Since the scanner algorithm is simple and clear, we will not describe it in this paper[3] and turn directly to the discussion of problems one may face while implementing a C++ parser.

[2]In general, the C++ grammar is not LL(1), and speaking of the recursive descent makes no sense. However, we may consider an algorithm similar to the recursive descent. The difference is that the deterministic decision what production should be chosen from the set of productions with the same left parts is made not only by the observed symbol (as it is in the recursive descent). The right context that can be unlimited in general is taken into account in addition. Such an algorithm can be called the *lookahead recursive descent*, because it also performs top-down parsing. Besides, the derivation tree of a chain is built implicitly by a set of recursive procedures, just like it is usually done by the usual recursive descent.

[3]The description of the scanner algorithm can be found in [6].

3.1. Left recursion issue

Our goal is to implement a parser that uses the lookahead recursive descent algorithm.[4] However, the C++ grammar in [3] is left-recursive; hence, it is impossible to implement the recursive descent. We have to transform the grammar so that it fits our needs. To achieve this, we use the transformation from [6] called the *change of direction of recursion*. Consider an example. The productions for the declarator in C++ have the following form:

```
declarator ::=
    dname
    modifier-list declarator
    ptr-operator declarator
    declarator ( argument-decl-list ) cv-qualifier-list opt
    declarator [ constant-expression opt ]
    ( declarator )
```

Let us recall how the *change of direction of recursion* looks like just to make the discussion more demonstrative. Let a grammar $\Gamma = (T, N, S, P)$ contain the following productions:

$$j : A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n, \tag{1}$$

where $A \in N$, $\alpha_1, \ldots, \alpha_m \in (T \cup N)^+$, $\beta_1, \ldots, \beta_n \in (T \cup N)^*$, and none of the chains β_i begins with the nonterminal symbol A. Consider the productions:

$$j_1 : A \to \beta_1 [B] \mid \ldots \mid \beta_n [B], \tag{2}$$

$$j_2 : B \to \alpha_1 [B] \mid \ldots \mid \alpha_m [B], \tag{3}$$

where $B \notin N$. Then the grammar $\Gamma_1 = (T, N \cup \{B\}, S, (P \setminus \{j\}) \cup \{j_1, j_2\})$ is equivalent to Γ. Let us apply this transformation to our productions. First, we introduce a new nonterminal symbol *declarator-tail*. Then we have:

```
declarator ::=
    dname declarator-tail opt
    modifier-list declarator declarator-tail opt
    ptr-operator declarator declarator-tail opt
```

[4] There are several reasons for implementing the lookahead recursive descent. First, there is no need to parse expressions in the same way it is done in compilation. Implementing an LR(2)-parser would therefore be expensive and irrational. Second, the recursive descent enables — without breaking its ideology and transformation of the language grammar — to combine procedures constructed according to grammar productions with those for simplified parsing of language constructions.

```
    ( declarator ) declarator-tail opt

declarator-tail ::=
    ( argument-decl-list ) cv-qualifier-list opt
    declarator-tail opt
    [ constant-expression opt ] declarator-tail opt
```

As one can see, the latter productions do not contain left recursion.

3.2. Grammar ambiguity

We should note a grammar ambiguity [3] with regard to the expression-statement and the declaration. Ambiguity resolving is purely syntactic, i. e., the meaning of a name (ignoring the difference between type name and other names) is not used for resolving. A practical rule for resolving ambiguities can be formulated as follows:

1. if something looks like a declaration, then it is a declaration, else
2. if something looks like an expression, then it is an expression, else
3. this is a syntax error.

Parsing with backtracking, together with this rule, may be used to resolve the ambiguities.

So, this rule resolves the ambiguities if differences between the type name and other names are taken into account. However, generally it is impossible to do this in the context of the function definition because types can be defined outside function bodies. Thus, there are constructs for which ambiguity cannot be resolved. Example:

```
id(a);    // if id is a type name, then it is a declaration,
          // else it is a function call
id1*id2;  // if id1 and id2 are not type names, then it is
          // an expression, else it is a declaration
```

For such constructs, ambiguity is resolved in favour of a declaration. An exception to this rule is the construct *scope-qualifier* $id(a)$, where *scope-qualifier* is a qualifier of the visibility scope and can be an empty string.

Note that parsing is still deterministic since the analyzer makes use of the right context to choose one of the following: the string should be parsed either as a declaration or as an expression. Thus, determinism of the analyzer is guaranteed. Moreover, if in the process of looking ahead the analyzer

decides that the next construct is a declaration, this means that it is actually parsed and can be skipped in the further analysis. In other words, an expression should sometimes be parsed twice while a declaration is parsed only once.

3.3. Notes on the primitive semantic analysis

According to the original statement of the problem (see Section 1), we have to determine what kind of semantic information the analyzer should collect. As we know, the task of the analyzer is to collect arguments and results, including the hierarchy of local variables of the subroutine. For this purpose, it is needed to perform the semantic analysis of declarators, blocks, and using occurrences of the variables.

The semantic analysis of declarators serves to determine whether a variable is a reference. To implement this analysis, we have to supply the declarator productions with some attributes. A simple consideration allows us to do this as follows:

productions	attribute rules
d1 ::=	
dname d-tail opt	dname.bRef = d1.bRef
m-list d2 d-tail opt	d2.bRef = d1.bRef
ptr-op d2 d-tail opt	d2.bRef = ptr-op.bRef
(*d2* **)** *d-tail* opt	d2.bRef = **false**
ptr-op ::=	
***** *cv-qualifier-list* opt	ptr-op.bRef = **false**
& *cv-qualifier-list* opt	ptr-op.bRef = **true**
scope-qualifier ***** *cv-q-list* opt	ptr-op.bRef = **false**
dname ::=	
name	name.bRef = dname.bRef
q-name	q-name.bRef = dname.bRef
name ::=	
id	id.bRef = name.bRef
~ *id*	
operator-function-name	
conversion-function-name	

The semantic analysis of blocks (visibility scopes) is needed to determine the visibility of local variables. A well-known approach to implementation of the analysis of visibility scopes is to use a stack of lists of objects visible in a given place in the subroutine. When the analyzer enters a new block, a new empty list is pushed into the stack. When the analyzer exits from the block, the stack pops the list. Using the stack, the analyzer distinguishes local and global variables in any place of the subroutine.

Finally, in the process of the analysis of using occurrences, the analyzer determines whether a subroutine uses or modifies a given variable. To implement this analysis, the analyzer extracts the context of the variable. It consists of four tokens on the right of the variable (practice has shown that four tokens are sufficient for the problem solution). This context is used to determine the type of the construction containing the variable as its leftmost token. Then the type of the construction allows the analyzer to compute the values of the attributes.

4. Java analyzer library

The next question is implementation of the analysis for the Java language (we remind that Java is considered in this section).

We will not implement a custom analyzer this time and try to make use of an existing one. However, we need some changes in its construction. To achieve this goal, let us use the Wasp analyzer since it performs a powerful static analysis and the Wasp source code is available.

So, there is the Wasp analyzer as a console application written in the mixture of Oberon2/Modula2. It is necessary to implement a Java library that exposes the Sap API interface and conforms the UI environment from the standpoint of data exchange format. There are at least two approaches to solving this problem. The first stands for calling the analysis procedures from Wasp object modules. In this approach, all the Wasp code is linked together with the library. Such a solution is motivated and plausible if Wasp is written in C/C++, because both the UI component and the Sap API are written in C++. In the second approach Wasp is used as a separate process using a file for data exchange. In this case, there is a library that starts Wasp as a separate process and receives the result of analysis in a file. This approach seems to be more realistic and should be chosen to solve the problem.

The summarization of the aforesaid ideas yields the following. It is still necessary to implement an algorithm for search of method headers in a module. At the same time, it is possible to use Wasp for the analysis of subroutines. This can be fulfilled by using the so-called *external analyzer scheme*.

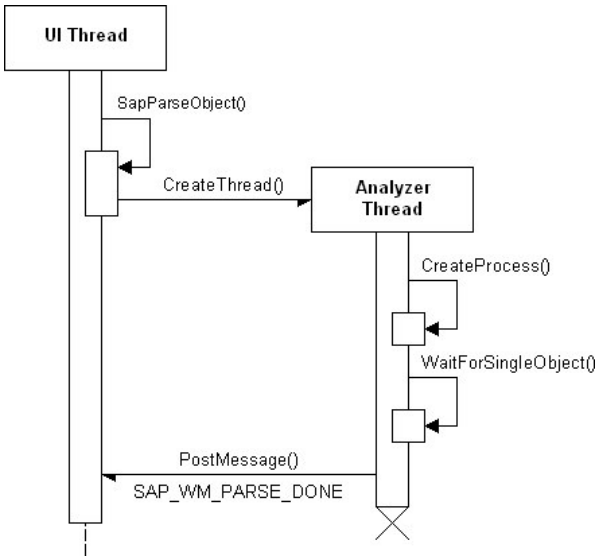

Figure 2. External analyzer scheme

4.1. External analyzer scheme

We assume that the interface method (let us call it *SapParseObject*) starting subroutine analysis should be implemented in accordance with the Sap API architecture. Figure 2 will help us to concentrate on the subject.

It shows a sequence of actions that *SapParseObject* should perform in order to analyse a subroutine. Note that we require the algorithm of analysis to run in a separate process. According to Section 2.4.1, *SapParseObject* creates a thread (Analyzer Thread in Figure 2) by using the *CreateThread* system function and returns control immediately to the user interface thread (UI Thread). The newly created thread performs the analysis of the subroutine. Whereas we want to use the external analyzer, *Analyzer Thread* runs Wasp in a new process by using a system call of *CreateProcess.* After this, *WaitForSingleObject* causes *Analyzer Thread* to sleep until Wasp finishes its work. After Wasp terminates, *Analyzer Thread* wakes up and calls the asynchronous *PostMessage* system function to post the SAP_WM_PARSE_DONE message to the UI Thread message loop. Then *Analyzer Thread* terminates.

Let us consider the analysis of subroutines containing syntax errors. In this situation, Wasp does not reach the stage of analysis. It outputs to the console an error message that should be displayed to the user and interrupts its execution. Sap API provides a list of errors to allow the UI environment to

display error messages. The address of the list is passed to the library when the interface functions are being called. Thus, it is necessary to intercept the Wasp console output as well as to form a list of errors before *PostMessage* is called. Besides, the Wasp format of error messages should be converted to that of Sap API.

4.2. Wasp — the XDS Java Static Analyzer

In this section, we will investigate necessary modifications of Wasp. They are obligatory since the direct use of Wasp is impossible in the context of the Sap architecture.

4.2.1. Modifications made to Wasp

Wasp obtains the desired FI (and even much more) and converts it into an internal representation. Traversing the internal representation and mapping FI to a special format (we will discuss this format later in detail) is nearly all that remains to be carried out.

We would like to underline that the Wasp internal representation contains a list of arguments and results for each class method. However, we also need a hierarchy of local variables that does not exist in the representation. Nevertheless, the representation provides us with a list of local variables. The list does not indicate whether a local variable is an argument or a result. In this case, the hierarchy of local variables is obtained by using the list of local variables and traversing the method's body. A different matter is to distribute local variables among arguments and results. In this case, the list of local variables and the list of arguments and results may be used. We apply a simple algorithm: for a given local variable, the list of arguments and results is searched through to determine whether the variable belongs to the list. The complexity of such an algorithm obviously is $\mathcal{O}\left(n^2\right)$, where $n = \max\left(l, k\right)$, l is the number of local variables in the method and k is the number of arguments and results of the method. This complexity is fully acceptable, taking into account that the average number of local variables, arguments, and results is not too large.

4.2.2. Exchange format

The format of data exchange between the Java library and Wasp has been chosen to be a subset of the standard vCard version 2.1 format that can be found in [11]. Let us discuss why such a choice has been made.

In fact, the format should reflect the hierarchy of local variables. The hierarchy is usually represented as a tree. Formally, we should build a one-to-one correspondence between the set of trees and a certain language. There-

fore, any context-free language that is able to express nesting fits our needs. One should use this opportunity to make the language simpler both for parsing and generating.

Since recently the XML language has become very popular. It fits conceptually our needs. However, using XML makes sense only if an existing parser is reused. At the same time, using an existing XML parser is not optimum in this situation. A better way is to use a subset of a standardized language to implement a constrained (simple) parser. A possibility to extend the subset to the entire language will thereby be provided. That is to say, a standard parser has to be used only when it is really needed. This is an advantage that enables an inexpensive extension of the analyzer library when moving to a next version. The vCard format just satisfies the mentioned conditions and may be used as a prototype for our purpose.

A subset of the selected language should now be obtained and some keywords replaced. A formal description of the format of data exchange is left out of the scope of this paper and can be found in Application B of [1].

5. Conclusion

In this paper, we propose an extensible facility intended to extract properties of the C++ and Java subroutines. This facility simplifies and accelerates the process of program understanding. It provides a convenient user interface and does not require much resources to operate. The description of user interface has been left out of the scope of this paper. However, one can easily learn them by reading the Sap on-line help system (see a reference below). The deterministic analyzer of the ANSI C++ function definition has been discussed. The analyzer implementation resides in a dynamic-link library enabling its late binding with the user interface. We have also studied modifications that should be introduced into Wasp in order to reuse it for the Sap processor. The format of data exchange has been described as a related issue.

The primary advantage of the proposed facility is believed to be the possibility of its extending, which has been verified for two languages. Therefore, creating analyzer libraries for new imperative languages might be a well-appreciated enhancement of Sap.

Finally, in order to make the discussion consistent, we provide the following information. Sap 1.0 and 2.1 can be downloaded from the web site `http://www.iis.nsk.su/cppsap`. The documentation on version 1.0 and the API for version 2.1 can be obtained from the same place.

Acknowledgements. The author would like to thank A. V. Zamulin for his constructive comments.

References

[1] Dubranovsky I. V. The Analysis of Information Links of Subroutines in C++ and Java. — Term Paper, Novosibirsk State University, 2001 (in Russian).

[2] Baburin D. E., Bulyonkov M. A., Emelianov P. G., Filatkina N. N. Visualization facilities in reverse engineering // Programmirovanie. — 2001. — № 2. — P. 21–23 (in Russian).

[3] Ellis M., Stroustrup B. The Annotated C++ Reference Manual. — AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

[4] Emelianov P. Analysis of the equality relation for the program terms // Proc. of the 3th Intern. Static Analysis Symposium. — Lect. Notes in Comput. Sci. — 1996. — Vol. 1145. — P. 174–188.

[5] Kasyanov V. N. Optimizing Program Transformations. — Moscow: Nauka, 1988 (in Russian).

[6] Kasyanov V. N., Pottosin I. V., Methods of Compiler Construction. — Novosibirsk: Nauka, 1986 (in Russian).

[7] Kuksenko S. V., Shelekhov V. I., The static source code checker of run-time errors // Programmirovanie. — 1998. — № 6. — P. 27–43 (in Russian).

[8] Naumovich G. N., Clarke L. A., Osterweil L. J. Verification of communication protocols using data flow analysis // Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Eng. — San Francisco, 1996. — P. 93–105.

[9] Pottosin I. V. A "good program": an attempt at an exact definition of the term // Programming and Computer Software. — 1997. — Vol. 23, № 2. — P. 59–69.

[10] Microsoft Developer Network Library. — October 1999.

[11] vCard — The Electronic Business Card Exchange Format. Version 2.1. — The Internet Mail Consortium (IMC), September 18, 1996 (`http://www.imc.org/pdi/vcard-21.doc`) plus the IrDA Telecom Extensions to the IMC vCard Format, Version 1.0, October 15, 1997 (p/o IrMC Specifications Package).