International Journal of Software Engineering
and Knowledge Engineering
Vol. 13, No. 1 (2003) 83–101
© World Scientific Publishing Company

A VISUAL SYSTEM SUPPORTING SOFTWARE REUSE IN THE BANKING LEGACY SYSTEM CONTEXT

GENNARO COSTAGLIOLA*, RITA FRANCESE† and GIUSEPPE SCANNIELLO‡

Dipartimento di Matematica e Informatica,
Università degli Studi di Salerno, Baronissi (SA), Italy
**gcostagliola@unisa.it*
†francese@unisa.it
‡gscanniello@unisa.it

Accepted 10 October 2002

Banking legacy systems intensively exchange messages in electronic format. Such systems are, for their nature, difficult to update and maintain. As a consequence, the introduction of new types of messages is a hard task. Adding new functionalities requires custom coding and software reuse is seen as a key to obtain a better time-to-market factor, risk and cost reduction. In this paper we describe the architecture and part of the implementation of the SI.RE. Visual System, an Interbanking Network Information System that lets us generate gateways for the exchange of electronic messages among banking legacy systems and supports software reuse. In particular, the SI.RE visual system contains a Visual Programming Environment that allows us to obtain a rapid development of the message handling functions. This environment implements a Visual Programming Language UVG that allows a programmer to reuse COBOL routines.

Keywords: Visual programming language; legacy system; visual programming environment; software reuse.

1. Introduction

A Legacy Information System (LIS) is a system hard to maintain and to update. LIS are widely spread and critical for the business they support. As a matter of fact, the information system has to reflect the continuous changes of business requirements. Over the years, LISs have had many maintenance interventions to satisfy new requirements and to change hardware and software configurations. As a consequence, they present a deteriorated software structure often characterized by replication of code. Because of the lack of documentation, the dependencies between the subparts of the system are often unknown and the overall understanding of the programs is very difficult [19, 26]. Thus, small modification often causes system failures that are hard to detect. It is well known that the applications of such systems are, for the most part, written in old programming languages, such as COBOL [5], FORTRAN, PL/1, RPG, and their hardware is obsolete. Besides, it is a matter of fact that legacy systems implement business rules not documented elsewhere, which could be hard and very time consuming to reconstruct.

For these reasons, in the last decade the Legacy Information System (LIS) problem has been largely investigated [7–10, 14, 19, 27–29]. It is widely recognized that LISs are too large and too critical to be substituted en masse because the system redevelopment (big bang approach or Cold Turkey) has a great risk failure and prohibitive costs [10]. Software reuse can help a company to gradually evolve their software. In fact, adding new functionalities requires custom coding and software reuse is seen as a key to obtain a better time-to-market factor, risk and cost reduction and to improve the software quality. By using this approach, the old software becomes the base for the new one and software reuse extends the system lifetime and let the entire system to be reused.

Several approaches have been proposed to integrate legacy programs also with Web Applications, see [3, 12, 23, 24] as an example. Moving to the Internet saves past investments: the system components implementing business functions and rules, and the database components can be wrapped and reused. Recently, there has been some work on migrating or integrating legacy systems into web infrastructures by using XML for legacy systems integration. In [24] XML is employed for encapsulating legacy COBOL programs.

The LIS problem is particularly felt in the banking context, characterized by a great diffusion of such systems. Banking Legacy Systems are business critical, are active 24 hours a day and intensively exchange messages in electronic format. The growing diffusion of the electronic payment systems, the need to respect precise transmission time and the general trend to eliminate paper documents force such kind of system to make large use of electronic messages. So far, the number and the type of messages are progressively growing and the legacy banking systems have been repeatedly modified and integrated to be able to make dialog with the external world. Such systems, for their legacy nature, resist modifications. As a consequence, the introduction of new types of messages is a hard task and requires custom coding.

In this paper we focus on the problem of exchanging new message types between banking legacy systems, each of them presenting its internal format. They communicate through the Interbanking Network that employs a proprietary communication protocol not based on the HTTP. Thus, to let such system communicate through this network, it is not possible to adopt Web technologies but it is necessary to convert an outgoing message having the format handled by the LIS application into the appropriate format of the interbanking network, and vice versa, for incoming messages. The aim of our work is to define a Visual System for Banking Legacy Systems that allows an application of a credit institute to dialog with the Interbanking Network without taking into account the translation of the messages. Thus, SI.RE provides an easy and semiautomatic way for generating, for each message, a COBOL program representing a gateway between the legacy banking system and the external world. The SI.RE visual system allows different user profiles to define, modify or display the structure of the elaboration processes associated to a particular electronic message. These operations have to be accomplished by allowing a rapid development and maintenance of the message handling functions that have to be easily integrated into the banking system. Visual programming languages can be naturally employed to this aim, allowing us to improve the productivity of an expert user and the code reusability. In this paper we show how a visual programming system — visual programming language and visual environment — has been integrated into the visual system SI.RE.

The visual language which we have studied and implemented, the User Variable Generator (UVG), belongs to the class of hybrid visual languages, i.e., UVG contains sentences with both textual and visual elements. It allows the programmer to describe COBOL routines visually. For this language we have implemented a complete User Variable Environment (UVE) with a user-friendly graphical interface. UVE lets the user concentrate his/her attention on the structure of a program and insert the COBOL instructions as textual annotation of the visual language symbols. This environment has been realized by using the grammar-based tool Visual Language Compiler-Compiler (VLCC) [13].

This paper has been organized as follows: Section 2 gives a description of the problem we have examined and provides an overview of how an event gateway works and its integration in a banking legacy system; Sec. 3 describes the visual system SI.RE architecture; Sec. 4 presents the gateway generation process; Sec. 5 gives an outline of the visual programming language adopted for adding new functionalities to banking legacy systems and the features available in the Visual Programming Environment which we have provided; Sec. 6 gives some details on the system implementation, such as how the field format correctness of a message is handled and how the UVE has been integrated inside the Visual System. Finally, we give the conclusions where we summarize our results.

2. The Event Gateway

Information flows between credit institutes through the Inter-Banking Network (IBN), supporting a proprietary message format following specific standards such as, for example, the SWIFT [25] inter-banking international standard or, in Italy, the SIA standard [22]. Banking legacy systems often have an internal message format different from the IBNs. Thus, an incoming (outgoing) message has to be translated into the host (IBN) format. Moreover, some banks communicate with the external world through a service center connected to the IBN. This center handles some network services, such as electronic receipts, and it can happen that some nodes have a system different from that of the service center.

As an example, if the service center receives an event, i.e. credit transfer, from the IBN directed towards a node connected to it, it translates the event format into the receiver's, signaling to the customer the event receipt with a Short Message Service (SMS) on the cellular phone, as shown in Fig. 1.

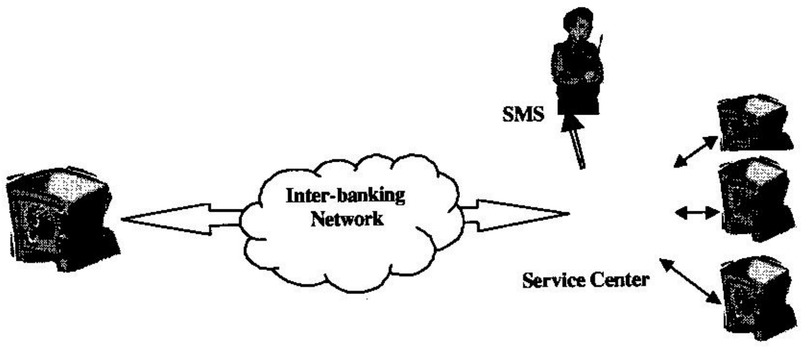


Fig. 1. The banking communication network.

Each time a new type of message is introduced, custom coding is required to operate the appropriate translation of the message formats.

The solution we propose allows an application of a credit institute to make dialog with the Interbanking network or with the service center without considering the translation of the messages. To this aim, a COBOL gateway program is generated which is capable of converting different type of data formats, or, more in general, events exchanged between legacy banking systems. For every input event, originated by a determined input channel, it is necessary to generate a specific Event Gateway Program. In this way, each credit institute is able to communicate with the external world by using its internal format that will be appropriately translated. An Event Gateway Program is called whenever that event is received and performs the following operations:

- checking that the input data are formally corrected;
- checking the applicative correctness. As an example, it verifies whether the banking account referred by the input event effectively exists;
- preparing the output events;
- handling errors.

Figure 2 shows the general architecture for handling an event originated by an input channel, such as Interbanking national network (RNI), the banking information system, SMS messages, UMTS message, etc.

Each Event Gateway Program is a COBOL routine that has to be integrated into the legacy banking system where, it will be compiled and put on-line. When the host receives an input event, it stores it into a table. As depicted in Fig. 3, the input event handler extracts the event to be served and calls the Event Gateway Program associated to the given input event type. The Event Gateway checks the correctness of the input event by calling the appropriate host routines and performs a series of operations, such as cyphering or dechyphering the event. Next, it builds the output event(s) and passes it (them) to an Output Event Handler. This program stores each output event into a table associated to the appropriate output channel.

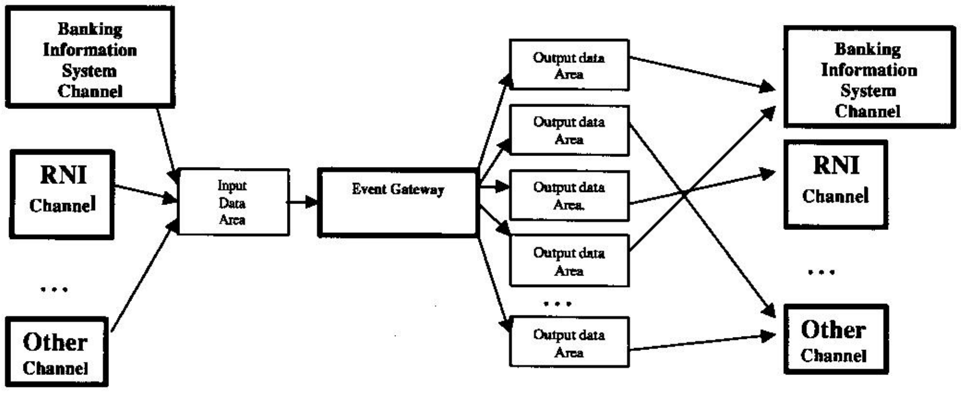


Fig. 2. The event handling scheme.

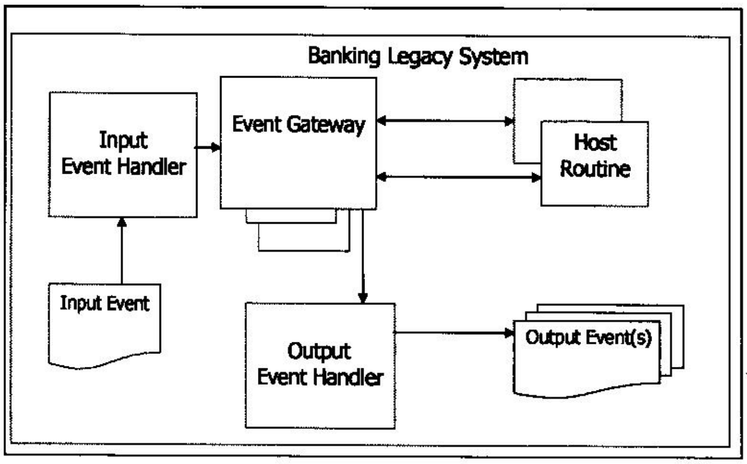


Fig. 3. The integration of the gateway program inside the banking legacy system.

In the literature, see [10] as an example, gateways provide a bridge between the legacy IS and the migration target environment. They are used for incrementally migrating the user interface, data or applications. In our approach the "gateway" term has a more restricted meaning: it is a COBOL program integrated into the Banking Legacy System translating a message into a BLS format or, vice versa, from the BLS format to another format. It allows the system to evolve, but it does not support system migration. Such kind of gateways already existed and programmers used to write them by hand. With our system, we generate gateways in a semi-automatic way for improving software quality and time-to-market factors.

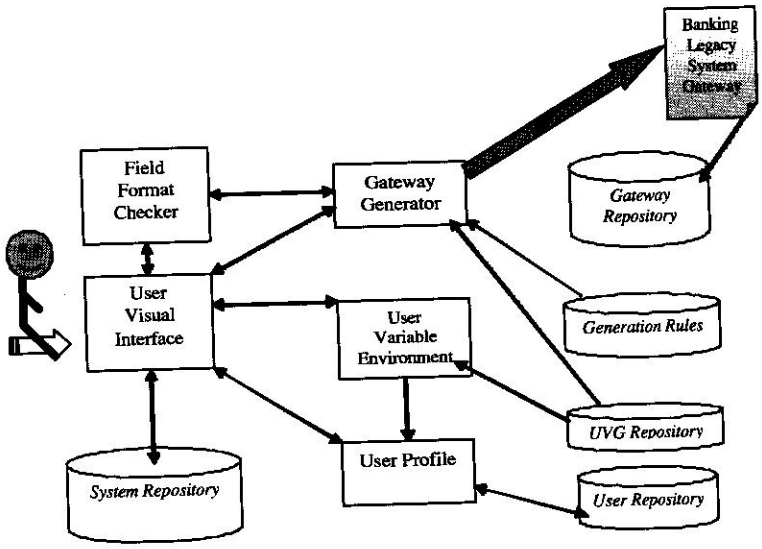


Fig. 4. The SI.RE Visual System architecture.

3. The SI.RE Architecture

Nowadays, there is an ever increasing pressure by banking institutes to let their applications be able to communicate with other external institutes. To this aim, the SI.RE visual system supports a large variety of message types and allows the users to create and customize their own data formats and to translate between format types. Thus, SI.RE eases and accelerates the development of Event Gateway Programs by supporting software reuse of existing COBOL code. Such a system has been realized by integrating heterogeneous components, as shown in Fig. 4.

The SI.RE Visual System information is stored into a repository constituted by five logical repositories:

- the *System Repository*, containing information associated to the events and to the COBOL routines to be reused. In particular, each event is characterized by a type (SIA, S.I. AMS, etc.) and by a layout consisting of an ordered sequence of fields. The COBOL routines are employed for error handling and event correctness check;
- the *User Repository*, containing the information about user access rights;
- the *Generation Rules*, containing information useful for the gateway generation, such as templates and code common to all the gateways;
- the *Gateway Repository*, containing the Event Gateway Programs generated by the SI.RE visual system;
- the *UVE Repository*, containing visual sentences expressed by the UVG language. These sentences are modifiable by using the User Variable Environment.

The other software components of the SI.RE architecture are:

- the *User Visual Interface* (UVI), the module allowing the user to handle the event data stored in the system repository; it assists the user in the definition and the generation of the event gateway program. Moreover, UVI allows the user to compose the output events in a very user-friendly way, starting from the given input event. Thus, it is possible not only to add some fields of the input event to the output event but also to create new fields;
- the *User Profile* module, handling the access rights to the visual system. We distinguish three professional profiles: the system administrator, handling the access permission; the designer, responsible for the gateway generation and of all the data associated; the generic user whose only permission is to display information;
- the *User Variable Environment* (UVE), an environment implementing the UVG visual language. It allows the designer to generate COBOL routines to be included into the event gateway program and will be better detailed in the following;
- the *Gateway generator*, a module that creates the event gateway program associated to a given input event. It receives the description of the input and output events from the Visual Interface together with the modality for compiling the output events.
- the *Field Format Checker* module, handling the lexical and syntactic correctness of event field formats during the gateway design.

4. The Gateway Generation Process

In this section, to better explain how the modules previously described interact, we detail the process for generating a gateway event program supported by the SI.RE visual system. An event gateway program is a COBOL routine that converts a message format into another message format and calls host COBOL routines to check the correctness of the input messages by considering both the formal and the applicative correctness. A gateway is created by following the steps of the process shown in Fig. 5.

Step 1. Input event definition: the Visual Interface Module enables the choice of the input event type. Actually, there exist several event types, such as SIA message, SMS message directed to or coming from cellular phones, UMTS (Universal Mobile Telecommunications System) message, proprietary types, etc. and the aim of our system is to easily extend both the event types and the number of events for a given event type (see Fig. 6). Once an event type has been chosen, it is possible to define the format of a new input event. Each event has associated to it the following information:

- the event code (i.e. '001'), a field that univocally identifies the event for a given event type;
- the event name, such as 'credit transfer';
- a direction: send or receive;

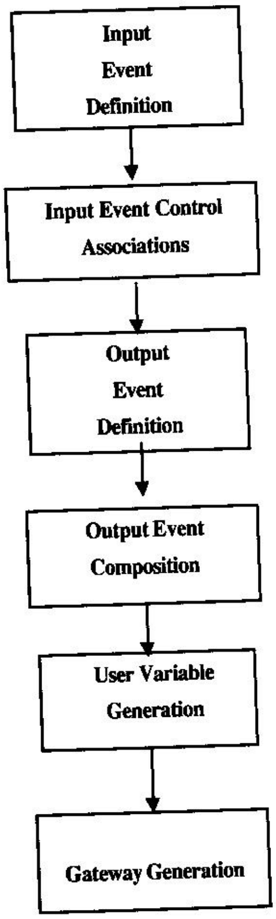


Fig. 5. The Gateway Generation Process.

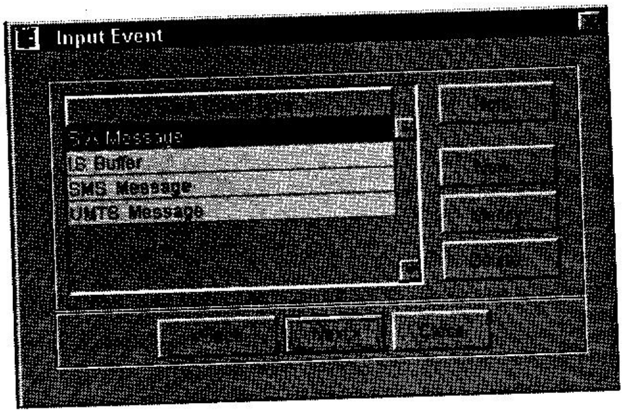
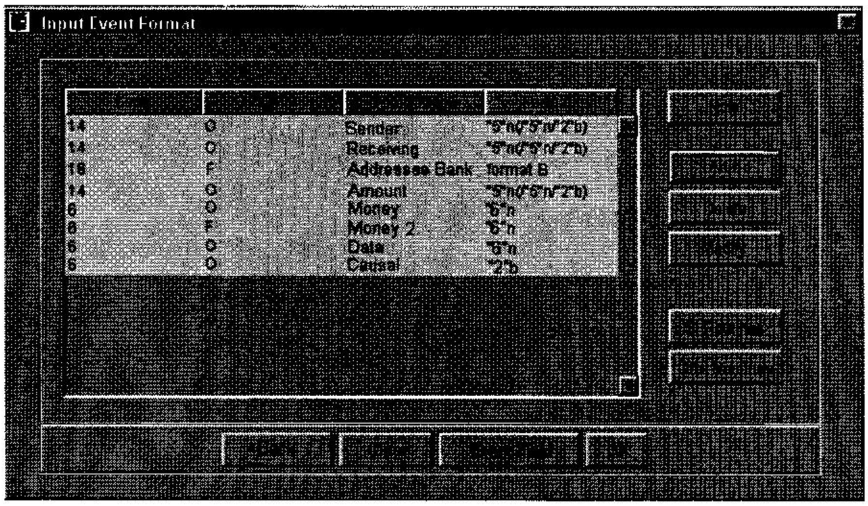


Fig. 6. Input Event Type.



14	O	Sender	"5"n("5"n("2"b)
14	O	Receiving	"5"n("5"n("2"b)
16	F	Addressee Bank	format B
14	O	Amount	"5"n("6"n("2"b)
6	O	Money	"6"n
6	F	Money 2	"6"n
6	O	Data	"6"n
6	O	Causal	"2"b

Fig. 7. An example of input Event Format.

- a description: note and normative reference;
- a message layout.

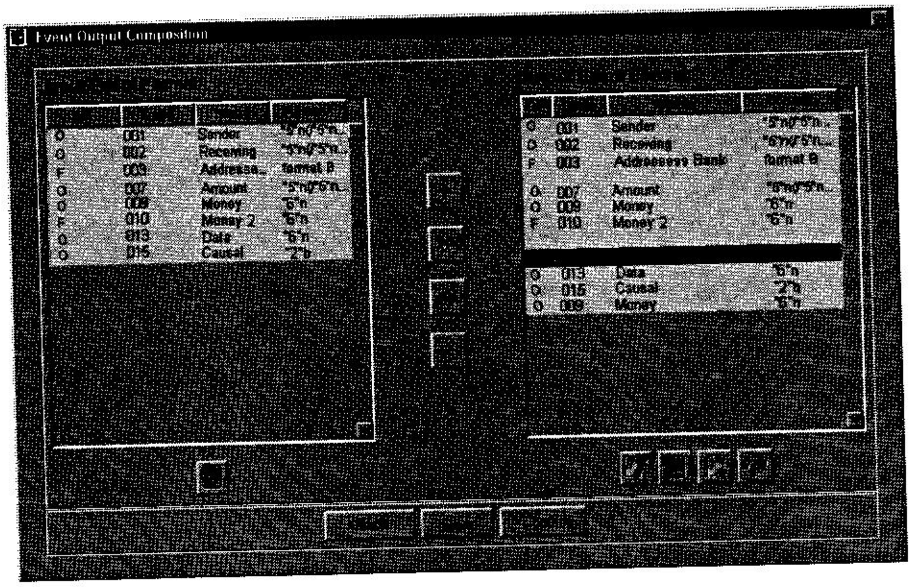
The message layout describes in detail the fields' succession composing the event. Each event field, as shown in Fig. 7, is described by specifying:

- the field name, such as "Sender";
- the field presence, i.e. facultative (F) or obligatory (O);
- the field format. It is specified by using a formalism very close to the regular expressions. Section 6 gives more details on the field format expressions and on the Field Format Checker module.

Step 2. Input Event Control Associations: during this step, the system proposes the catalog of the routines, stored it into the system repository for controlling the input message correctness. As an example, if a facultative field is present, the presence of another facultative field can become mandatory and has to be checked.

Step 3. Output Event Definition: the output event(s) associated to the given input is (are) defined. For this (these) event(s) it is necessary to specify the same information provided for the input event (type, code, description, etc.). As an example, we can associate two output events to the credit transfer event: an SMS message to the customer cellular phone (type= SMS, code = 01200, etc.) and a message to the BLS (type = IS buffer, code = A302, etc.).

Step 4. Output Event Composition: the layout of an output event is obtained by elaborating the layout of the input one. During this step, the fields that remain unaltered are exported from the input event layout. It is possible to place a field into a different position and some fields can be left blank. See Fig. 8 as an example.



O	001	Sender	"5"n0"5"n...
O	002	Receiving	"5"n0"5"n...
F	003	Addresse...	format B
O	007	Amount	"5"n0"6"n...
O	009	Money	"6"n
F	010	Money 2	"6"n
O	013	Date	"6"n
O	015	Causal	"2"n

O	001	Sender	"5"n0"5"n...
O	002	Receiving	"6"n0"5"n...
F	003	Addressees Bank	format B
O	007	Amount	"8"n0"5"n...
O	009	Money	"6"n
F	010	Money 2	"6"n
O	013	Data	"6"n
O	015	Causal	"2"n
O	009	Money	"6"n

Fig. 8. Composition of the output event format.

For automatically generating the output event, it is necessary to specify how its field has to be filled. To this aim, we associate to each field one of the following assignment modality:

- direct, i.e. the same value as an input event field;
- constant, i.e. a constant value;
- system, i.e. a system value such as the current date;
- a user variable, a value obtained elaborating the input data. A user variable is a COBOL procedure that calculates the value of the given output field by using the input event data. As an example, the input field named "data" in Fig. 7 is unpacked in the "day", "month", "year" fields of the output event.

Step 5. User Variable Generation: if the User Variable modality of assignment has been chosen, the systems runs the User Variable Environment for the generation of the COBOL routines that fill the output event fields having a format different from the input one. The next section better details these aspects.

Step 6. Gateway Generation: once the assignment modalities have been provided, the Gateway Generator module creates the Gateway Program by using the information about the input and output events for structuring the COBOL record layouts into the WORKING-STORAGE SECTION. It also includes the COBOL code generated by the User Variable Environment for filling the output fields that cannot be filled by a simple MOVE instruction.

5. The User Variable Environment

In this section we present the functionalities of the User Variable Environment for the UVG language. First, we provide a short overview of some basic concepts about visual programming languages and next, we give the basic characteristics of the UVG language and its implementation through the UVE.

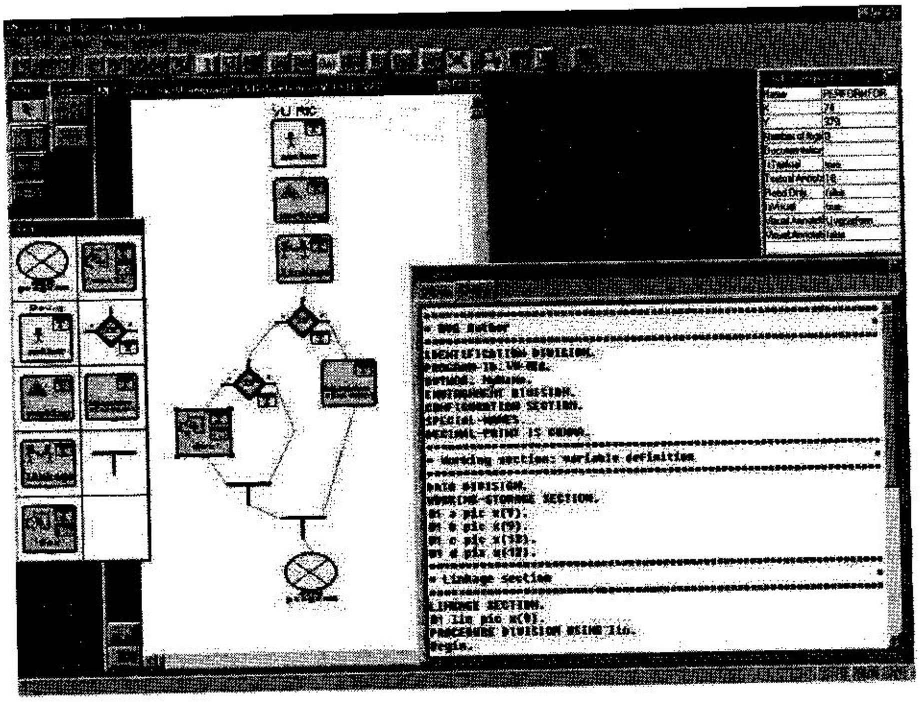
A *visual language* may be conceived as a collection of visual sentences composed by graphical objects arranged in two or more dimensions. Visual Languages are syntactically described by the grapical objects of the language, the relation used to compose the sentences, and a set of rules defining the set of visual sentence belonging to the language. Graphical attributes characterize the object image. Syntatic attributes are used to relate graphical objects in order to form visual sentences. A set of graphical objects form a visual sentence once all the syntactic attributes have been instantiated. A visual sentence is correct if it is lexically and syntactically correct [15].

VPLs are programming languages where visual techniques are adopted to express relationships among or transformations to data [6,15]. Nevertheless, practical experience has shown that visual programming requires the use of text, because pictures allow to define too high level operations and the most complex details can be specified only through texts [11]. In particular, the text is necessary for describing documentation, distinguishing between elements that are of the same kind, and expressing algebraic formulas [6].

The User Variable Generator (UVG) visual language has been adopted for generating COBOL routines and for supporting the reuse of the existing ones.

UVG is a hybrid language, i.e. it integrates visual and textual notation. In particular, it implements a type of flowchart suitable for the COBOL programming, as shown in Fig. 9, presenting a screen dump of the User Variable Environment where an UVG sentence has been drawn and the corresponding COBOL code has been generated. The graphical aspect of UVG symbols is shown on the symbol palette available in the User Variable Environment, on the left of Fig. 9. The visual sentences of UVG are obtained by connecting the symbols through links, as an example see the sentence depicted in Fig. 9, and by specifying the symbol annotations. The annotation types supported by the UVE are the following:

- the **visual annotation**, i.e., the possibility of associating a given symbol with a visual sentence belonging to another language. Such a facility is useful for implementing sub-processes and for supporting structured programming. In particular, we visually annotate the symbol STAT with another flow-chart language (UVG Perform language). The *UVG Perform language* is a modified subset of UVG useful for specifying visual annotation. Such a language presents the same connection rules as the UVG ones, the ***author***, ***working*** and ***linkage*** symbols are not present, being this language employed for subroutines, the ***end program*** symbol is replaced by the ***end perform*** and the subroutine begins with a ***begin perform*** symbol. In this way, we support structured visual programming.



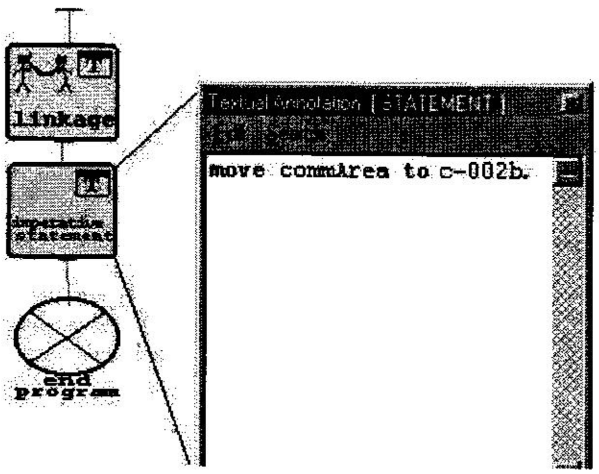
```
*****************************************************************
* [illegible] Author                                            *
*****************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. [illegible].
AUTHOR. [illegible].
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
DECIMAL-POINT IS COMMA.
*****************************************************************
* Working section: variable definition                          *
*****************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01 a pic x(9).
01 b pic x(9).
01 c pic x(12).
01 d pic x(12).
*****************************************************************
* Linkage section                                               *
*****************************************************************
LINKAGE SECTION.
01 lin pic x(9).
PROCEDURE DIVISION USING lin.
begin.
```

Fig. 9. A snapshot of an UVG sentence and the corresponding generated COBOL code.

- the **textual annotation**, i.e., the possibility of associating text with a given symbol, as shown in Fig. 10 where a textual annotation for the symbol "imperative statement" is depicted. To check the correctness of such an association, the annotated text is subject to a textual scanning and parsing. To this aim, different lexical and syntactic analyzers correspond to each symbol. In fact, each symbol represents a subset of the COBOL language or English-like strings. The textual scanner has been generated by lex, whereas the parser by yacc. One of the main uses of lex is as a companion to the yacc parser-generator, ensuring lexical and syntactic correctness of textual annotation. As an example, both the conditional symbol and the for loop are annotated by a text. The text associated to the former describes the condition, such as "A IS NOT ZERO", where A is a variable and the one associated to the latter consists of the specification of the iteration number of the annotated visual sentence.
- the **direct textual annotation**, i.e. the possibility of associating some text by writing directly on the symbol.

5.1. *The UVE output*

The visual sentence is compiled by clicking on the lightening button on the top menu bar. If the visual sentence and the annotated texts are error free, the compiler



```
move commArea to c-002b.
```

Fig. 10. An example of textual annotation.

generates a lexically, syntactically and semantically correct COBOL routine. When a symbol contains incorrect COBOL code, the compiler highlights it and displays a window showing the number of the line where the error occurs. Figure 9 depicts a correct UVG visual sentence and the COBOL code generated by compiling it.

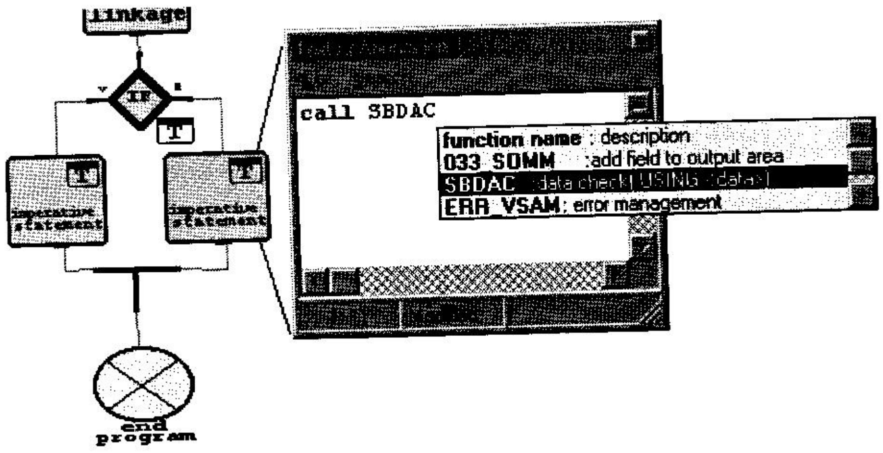
5.2. *Reusing COBOL code*

The UVE supports the reuse of code components such as the existing COBOL routines and the routines generated by UVG. As shown in Fig. 11, during the visual sentence creation it is possible to access the System Repository and select the appropriate routine to be called in the considered statement block. A user can find a routine by selecting the routine name in a list of available function names. Once a routine has been chosen, it is possible to obtain examples showing the results of its applications and how to use it.

For populating the code repository it is necessary to select a suitable code to reuse and to include it into a library [13]. The routines stored here are, for the most part, standard routines provided by IBM. The others are specific of the banking legacy system and have to be extracted from it, if they are not available, by using some reengineering technology for acquiring reusable assets, see [4] for an example.

6. Implementation Issues

In this section we outline some details regarding the system implementation. In particular, we examine the way we handle the field format correctness and the integration of the User Variable Environment inside the Visual System.



```
call SBDAC
```

```
function name : description
033 SOMM    :add field to output area
SBDAC  :data check[ USING <data>]
ERR VSAM; error management
```

Fig. 11. Textual annotation and code proposal.

6.1. *The field format correctness*

As previously described, during the gateway generation process it is necessary to define the event layout. Each event field has a specific format represented by a *field format expression*, a pattern of text that consists of alphabetic, numeric and special characters known as metacharacters. This pattern describes the format to be respected by the event field and that has to follow the rules depicted in Table I.

Table 1.

- length and information field:
  nn: max length (1 to *nn* characters)
  nn – *mm*: min and max length
  "*nn*": fixated length
- admitted characters:
  n: only numeric
  b: numeric and/or alphabetic characters
  x: all ASCII characters less the : and /. Received unexpected characters produce a transmission error.
  a: alphabetic characters.

The metacharacter "/" specifies a concatenation operation. As an example, 15n/1a indicates that fifteen numeric characters have to be followed by the "/" special character and one alphabetic character.

The expression enclosed in parenthesis is optional, i.e. the field format expression "5"n(/"5"n/"2"b) is formed by either five numeric characters or five numeric characters followed by "/", five numeric characters, "/" and two numeric and/or alphabetic characters. It is possible to have several nested level of parentheses.

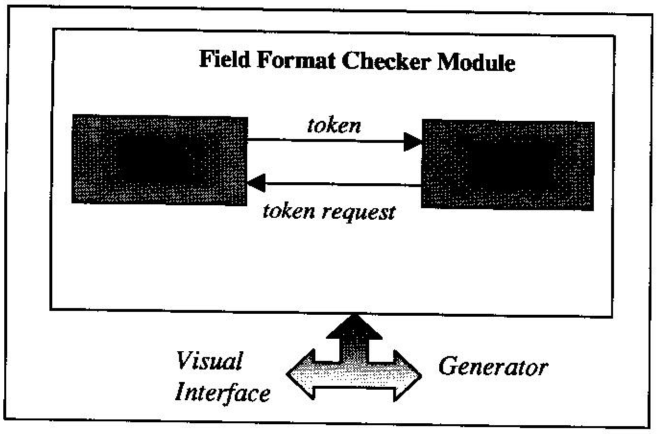


Fig. 12. The Field Format Checker Module.

This type of expression has to be formally described by a deterministic context-free grammar. The characteristics of this grammar, see [1] for more details, allow us to employ a LALR(1) parser for the recognition of format field expressions.

In practice, the correctness of a given field format is handled by the Field Format Checker module that employs syntactic and semantic analyzers, respectively, the lexer and the parser, as shown in Fig. 12, created by using the Java-Lex and Java-Cup tools [14]. In particular, Java-Lex, in a way very similar to Lex, receives an input file containing the specification of a lexical analyzer and generates the required lexical analyzer written in Java. Java-CUP (Constructor of Useful Parser), in a way very similar to YACC, is employed for generating LALR parsers for Java. The lexer for the field format expression has been generated by providing the lexical analyzer specification to Java-lex. Once the lexer has been compiled, it is able to analyze the input string and extract tokens from it. On the other hand, the parser has been generated by providing the grammar specification in input to Java-CUP and by compiling its output with a Java compiler.

As depicted in Fig. 12, the Field Format Checker Module (FFCM) receives input by the Visual Interface the format field expression associated to a given field of the input event. The parser checks the syntactic correctness requiring tokens from the lexer, by invoking the yylex function. If the expression is not correct an exception occurs that will be captured and handled by the VI module. Otherwise, the FFCM module provides information concerning the field format to the Generator. This information let the generator declare in the WORKING-STORAGE SECTION the COBOL layout of the record associated to the given field of the input event. Figure 13 shows how the generator constructs the layout of the 'sender' field, "5"n(/"5"n/"2"b).

```
000255*****************************************************************
000260*                          Record : Sender                         *
000265*****************************************************************
000270*****************************************************************
000275*                Description : Banking Sender Code                *
000280*****************************************************************
000285      03 WORK-AREA-001.
000290        05 C-5N-004-001-1 PIC 9(5) VALUE ZERO.
000295        05 WORK-SLASH-1 PIC X(0001) VALUE '/'.
000300        05 C-5N-004-001-2 PIC 9(5) VALUE ZERO.
000305        05 WORK-SLASH-2 PIC X(0001) VALUE '/'.
000310        05 C-2B-004-001-3 PIC X(2) VALUE SPACES.
```

Fig. 13. The 'sender' field declaration.

6.2. *Visual programming environment and visual programming environment generators*

It is common opinion that to effectively use a visual language it is necessary to have a Visual Programming Environment supporting the given language and enabling the editing and the compilation of a visual sentence in the given language [15]. The User Variable COBOL routines are genereated with the support of the User Variable Environment providing a set of visual symbols and relationships for drawing the visual sentence and to compile UVG sentences. To design and implement UVE we have been supported by the Visual Language Compiler Compiler (VLCC) system. VLCC is a grammar-based graphical system that inherits and extends to the visual field concepts and techniques of compiler generation tool like YACC [13]. In general, VLCC offers the opportunity of creating a visual environment for a specific visual language. It takes as input the syntax and the semantics of a visual language and the graphical aspects of its components and generates an integrated environment composed by a graphical editor and a compiler for the implemented language. Other Visual Programming Environment generators can be found in the literature, see, for example [20, 30].

6.3. *The native UVE*

The User Variable Environment has been developed by generating a C++ module with the VLCC system and has been integrated into the Visual System, implemented by JAVA. For integrating the UVE native code into the Visual System Java application the *Java Native Interface* (JNI) has been employed [21]. JNI let a Java application interact with modules written in a language other than Java. To this aim, the keyword *native* inside a Java program declares native code methods. Figure 14 shows the declaration of the native method LaunchUVE, running the User Variable Environment.

A native method declaration can foresee a return value and does not contain Java code. Once a native method has been declared, the Java program is compiled

```
public class NativeUVE {
public native boolean LaunchUVE( String vlcc, String pathVcm, int
protection, String sentence ,String cfile);

    static {
            System.loadLibrary("launcher");
            }
}
```

Fig. 14. The Java native method declaration for LaunchUVE.

as shown below, i.e. as any other non-JNI Java program.

```
javac NativeUVE.java
```

The javah utility creates a header file that can be used as a guide during the implementation of the native code method. The following command line creates the interface, shown in Fig. 14, between the NativeUVE in C++ and the Visual System.

```
javah -jni NativeUVE
```

7. Conclusions

This paper addresses a specific problem of the banking context that has a major relevance in the market of software and services, but it is traditionally closed to innovative solutions. In particular, visual languages and advanced interfaces have a very limited spread. This is mostly true for Banking Legacy Systems. These systems have to support intensive message exchange made even more difficult by the fact that each one of them has its own message format. Moreover, they must handle the introduction of new message types, requiring custom coding for appropriate translation of the new messages in both reception and transmission. To address these problems we have presented the SI.RE Visual System. It easily generates COBOL programs for handling new messages and reusing COBOL software routines of the legacy system. SI.RE allows us to define and convert different types of data formats. In the paper we have described, in particular, the gateway generation process for creating COBOL programs and the User Variable Environment (UVE) integrated into the SI.RE visual system whose aim is to obtain a rapid development of the new message handling functions. UVE implements a Visual Programming Language UVG that supports reuse of COBOL routines. The SI.RE Visual System integrates the routines generated by UVG and the existing routines of the Banking Legacy System. Thus, the generated gateway is a COBOL routine interacting with the BLS and calling its COBOL modules.

The approach relies on the Visual Language Compiler-Compiler (VLCC) System previously developed by the authors for the generation of visual environments. At present, a Java prototype of the visual system SI.RE is running on a PC/Windows

'98. The Visual Environment for UVG has been developed by generating a C++ external module with the VLCC and has been integrated into the Visual System.

In the future, we plan to investigate how to structure the routine repository for a more effective reuse by also using reverse engineering techniques, such as the ones suggested in [2], and how to extend the system to other language types. Moreover, we are interested in investigating how to define and implement a visual language suitable for users who are not expert in COBOL and how to allow a system user to define several types of routines, such as applicative checking, error handling, etc., by using UVE.

Acknowledgements

The authors would like to thank the industrial partner, Gruppo Net S.p.A.-Rome, for the support provided.

References

1. R. S. Aho and J. Ullman, *Compilers*, Addison-Wesley, 1988.
2. J. D. Ahrens and N. S. Prywes, "Transition to a legacy- and reuse-based software life cycle", *IEEE Computer* **28**(10) (1995) 27-36.
3. L. Aversano, A. Cimitile, G. Canfora, and A. De Lucia, "Migrating legacy systems to the web: An experience report", in *Proc. European Conf. on Software Maintenance and Rengineering*, Lisbon, Portugal, 2001, IEEE Comp. Soc. Press 2001, pp. 148–157.
4. R. Arnold and W. Frakes, "Software reuse and reengineering", *CASE Trends*, Feb. 1992.
5. E. C. Arranga and W. Price, "Fresh from Y2K, What's Next for Cobol?", *IEEE Software*, March/April 2000.
6. M. Baker and J. Power, "Visual Programming Languages", *cse505*, December 1994, http://www.cs.washington.edu/homes/jpower/vpl/vpl_home.html
7. K. Bennet (guest editor), "Special Issue on Legacy Systems", *IEEE Software* **12**, no. 1 (1995).
8. A. Bianchi, G. Costagliola, P. D'Ambrosio, R. Francese and G. Scanniello, "A visual system for the generation of banking legacy system gateways", in *Proc. 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, Stresa, September 5–7, 2001, pp. 350–357.
9. J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy Information Systems: Issues and directions", *IEEE Software* **16**, no. 5, Sept.–Oct. 1999.
10. M. L. Brodie and M. Stonebraker, *Migrating Legacy Systems*, Morgan Kaufmann, Inc., 1995.
11. M. M. Burnett, A. Goldberg and T. G. Lewis, *Visual Object-Oriented Programming*, Manning Publications Co., 1995.
12. G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, "Decomposing legacy programs: A first step towards migrating to client-server platforms", *The Journal of Systems and Software*, **54** (2000) 99–110.
13. G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia, "Automatic generation of visual programming environments", *IEEE Computers*, March 1995, pp. 56–66.
14. F. P. Coyle, "Legacy integration — changing perspectives", *IEEE Software*, March/April 2000.

15. F. Ferrucci, G. Tortora and G. Vitiello, "Visual programming", *Wiley Encyclopedia of Software Engineering*, ed. J. J. Marciniak, 2nd Edition., December 2001.
16. G. C. Gannod, Y. Chen and B. Cheng, "An automated approach for supporting software reuse via reverse engineering", in *Proc. 13th IEEE Int. Conf. on Automated Software Engineering*, 1998, pp. 94–103.
17. Java-Lex and Java-cap tutorial.
http://www.hio.hen.nl/~vanleeuw/cano/tutorial.html#LEX.
18. K. C. Kang, S. Kim, J. Lee and K. Lee, "Feature-oriented engineering of PBX software for adaptability and reusability", *Software-Practice and Experience* **29**(10) (1999) 875–896.
19. A. Lauder and S. Kent, "Legacy system anti-pattern-oriented migration response", ed. P. Henderson, *Systems Engineering for Business Process Change*, Springer Verlag, 2000.
20. M. Minas and G. Viehstaedt, "Diagen: A generator for diagram editor providing direct manipulation and execution of diagrams", in *Proc. 11th IEEE Symp. Visual Language (VL'95)*, Sept. 1995, pp. 203–210.
21. D. Parson and Z. Zhu, "Java Native Interface idioms for C++ class hierarchies", *Software — Practice and Experience* **30**, no. 15 (2000) 1641–1660.
22. SIA S.p.A., Interbanking Society for Automation. www.SIA.it.
23. M. A. Serrano, "Evolutionary migration of legacy systems to an object-based distributed environment", in *Proc. IEEE Int. Conf. on Software Maintenance*, 1998.
24. H. M. Sneed, "Wrapping Legacy COBOL Programs behind an XML-Interface", in *Proc. Eighth Working Conf. on Reverse Engineering (WCRE '01)*, 2001.
25. http://www.swift.com/
26. P. Tonella, G. Antoniol, R. Fiutem and F. Calzolari, "Reverse engineering 4.7 million lines of code", *Software — Practice and Experience*, no. 30 (2000) 129–150.
27. A. Umar, *Application (Re)engineering: Building Web-Based Applications and Dealing With Legacies*, Prentice Hall, June 1997.
28. I. Warren, *The Renaissance of legacy systems: Method support for Software System Evolution*, Pratictioner Series, Springer, March 1999.
29. N. Weiderman, L. Northrop, D. Smith, S. Tilley, and K. Wallnau, "Implications of Distributed Object Technology for Reengineering (CMU/SEI-97-TR-005)", —
http://www.sei.cmu.edu/publications/documents/97.reports/97tr005/97tr005abstract-.html.
30. K. Zhang, D Zhang and J. Cao, "Design, construction, and application of a generic visual language generation environment", *IEEE Trans. on Software Engineering* **27**, no. 4 (2001) 24–34.