Computer Languages, Systems & Structures 54 (2018) 20–38

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Using the local context for the definition and implementation of visual languages

Gennaro Costagliola, Mattia De Rosa*, Vittorio Fuccella

Department of Informatics, University of Salerno, Via Giovanni Paolo II, Fisciano 84084, SA, Italy

ARTICLE INFO

Article history:
Received 13 February 2018
Revised 28 March 2018
Accepted 3 April 2018
Available online 12 April 2018

Keywords:
Local context
Visual languages

ABSTRACT

In general, visual languages need to be simple in order to be easily used and understood. As a result, many of them have simple constructs that can be defined by simply describing local constraints on the constituent elements. Based on this assumption, in a previous research, we developed a local context methodology for the specification of the syntax of simple visual languages such as flowcharts, entity-relationship diagrams, use-case diagrams. In this paper, we extend the methodology by defining a new technique for a local context based semantic translation of a visual language. The technique uses XPath-like expressions, called SGPath, together with a data flow model of execution. As for the case of local syntax checks, attributes and rules to calculate them are defined for each element of the language. For a given element in the abstract sentence graph, the SGPath expressions are used to gather values from its neighbors in order to allow the rules to calculate its semantic attributes. The new methodology has been implemented in the tool LoCoMoTiVe and been tested on visual languages such as entity-relationship diagrams, flowcharts, trees.

1. Introduction

Since their introduction, visual languages have been defined as part of systems which enhance communication by the use of visual elements. Diagrams, maps, images and pictures are examples of visual sentences and they are used as representations of mental concepts that need spatial contexts in order to be described naturally. Their role is to facilitate the communication among people since, when adequate, visual communication is more direct and of immediate understanding compared to the verbal or text type of communication. Because of this, the use of visual languages can be found almost in any context ranging from art to engineering.

It must be admitted, however, that badly designed visual languages can lead to visual formulations that are very difficult to compose and interpret. In this case, they fail their main reason of being.

In general, from a syntactic point of view, this occurs when a language presents many syntactic rules binding elements which can be very far in a sentence. As an example, in the case of textual programming languages, one might consider the matching parenthesis in languages such as C or Java.

To overcome this problem, block visual languages such as Scratch have eliminated syntax dependencies between far language elements, by adding shape information to each element and reducing the composition of a program to the creation of a puzzle. The syntactic rule here is very simple: “a visual program is syntactically correct if and only if each tile well

* Corresponding author.
E-mail address: matderosa@unisa.it (M. De Rosa).

https://doi.org/10.1016/j.cl.2018.04.002
1477-8424/

interlocks with its neighbors". In this case, the validity of the local shape constraints on each tile guarantees the correctness of the whole visual program independently on how many elements it is made of.

This is also why block languages are now very popular and extensively used in education for teaching introductory programming to non-experts, [1].

In previous research, [2,3], we have shown that not only block languages but also many other very well known and used programming visual languages such as unstructured flowcharts, data flow languages and entity-relationship diagrams can be syntactically specified by mostly setting local constraints on the language elements. This has allowed us to avoid writing complex grammars for these types of languages by greatly simplifying the design of a visual language from a syntactic point of view.

In particular, our methodology, known as *local context-based visual language specification*, only requires the language designer to define the *local context* of each symbol of the language. The local context is seen as the interface that a symbol exposes to the rest of the sentence and consists of a set of attributes defining the local constraints that need to be considered for the correct use of the symbol.

In this paper, we continue our previous work by facing the semantic translation of a visual language based on the local context. We propose a method based on XPath-like expressions, called SGPath expressions, and a data flow model of execution in order to define the semantic translation rules for the language by specifying rules for each single language element as opposed to defining semantic rules for complete phrases.

In particular, for a given node of the abstract syntax graph returned by the syntactic phase, the SGPath expressions are exploited to gather values from its neighbors to be used in the translation. XPath-like languages have shown to be very well-suited for navigation through graphs when querying data held in the nodes, [4].

The new methodology has been implemented as part of the tool LoCoMoTiVe [3] and tested in two different case studies, aimed at generating code from two different visual languages: C-like code from flowcharts and SQL code from entity-relationship diagrams.

The paper is organized as follows: the next section refers to the related work; Section 3 recalls the main concepts of the local context specification of visual languages; Section 4 describes the SGPath specification; Section 5 describes the local context-based semantic definition (LCDS) and its analysis algorithm; Section 6 describes the LoCoMoTiVE tool that implements the methodology; finally Section 7 concludes the paper with final remarks and a brief discussion on future work.

2. Related work

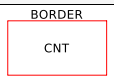
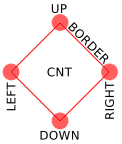
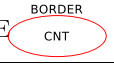
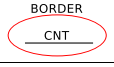
Several strategies to model diagrams as visual language sentences have been conceived in the past. Diagrams have been represented either as sets of attributed symbols, with the "position" of the symbol in the sentence represented through typed attributes (*attribute-based approach*) [5], or as sets of relations on symbols (*relation-based approach*) [6]. Despite the fact that the two approaches appear to be different, they both consider a visual sentence as a set of symbols and relations among them. This structure can be represented as a spatial-relationship graph [7] built by adding a node for each graphical symbol and adding an edge for each spatial relationship between symbols (nodes).

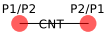
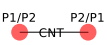
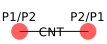
In the attribute-based approach, the relations are derived by matching attribute values, while in the relation-based approach the relations are explicitly named.

Based on these representations, various formalisms have been suggested to represent the visual language syntax, each one associated to ad-hoc scanning and parsing techniques: (Extended) Positional Grammars [8], Constrained Set Grammars [9], Relational Grammars [10], Reserved Graph Grammars [11] to name some (other approaches and details can be found in [12] and [13]). These visual grammars are defined, in general, by specifying an alphabet of graphical symbols together with their "visual" appearance, a set of spatial or topological relationships, and a set of grammar rules, usually in a context-free like format even though their descriptive power is mostly context sensitive.

Various software tools for visual language prototyping have been designed and implemented based on the different types of visual grammar formalisms. These include: VLDesk that is based on positional grammars [14], GenGed that is based on Hypergrah grammars [15], Penguin that is based on constraint logic based grammars[16], DiaGen [17] that is based on Hypergrah grammars, VisPro that is based on Reserved Graph Grammars [18], AToM3 [19], VL-Eli [20] and its improvement DEViL [21]. Most of the systems include the possibility to add visual language semantics specifications but these are all specified at the level of grammar productions (see [22] as an example). DEViL also provides an object-oriented like domain specific language and a library of "visual patterns" that can be used to easily specify common concepts such as lists, sets, tables, trees, etc. However, our work goes a step further by completely removing the grammar specification.

Even though context-free like rules are well known, visual grammars are usually difficult to define and read. This may be the reason why there has been not much success for these techniques in real-world applications. Many visual languages used today are syntactically simple languages that focus on the basic graphical elements and their expressive power, and therefore they do not need complex grammar rules to be specified. Because of this, we feel that our methodology allows a simpler specification for many of them, making it less demanding to define and quickly prototype visual languages with their semantic translation.

Symbol name	Graphics	Symbol occurrence	Attaching areas name	type	constraints
ENTITY		≥ 1	*BORDER*	EntBrd	$connectNum \geq 0$
			CNT	***text:***EntName	*string*
RELATION		≥ 0	*UP*	RelVrt	$\sum(connectNum) = 2$
			LEFT	RelVrt	
			RIGHT	RelVrt	
			DOWN	RelVrt	
			BORDER	RelBrd	$connectNum \geq 0$
			CNT	***text:***RelName	*string*
ATTRIBUTE		≥ 0	*BORDER*	AttBrd	$connectNum = 1$
			CNT	***text:***AttrName	*string*
KEY_ATTR		≥ 0	*BORDER*	AttBrd	$connectNum = 1$
			CNT	***text:***KeyName	*string*

Connector name	Graphics	Connector occurrence	Attaching areas name	type	constraints
CON_E_R		≥ 0	*P1*	EntBrd	$connectNum = 1$
			P2	RelVrt	$connectNum = 1$
			CNT	***text:***Cardin	*([01],[1N])*
CON_E_A		≥ 0	*P1*	EntBrd	$connectNum = 1$
			P2	AttBrd	$connectNum = 1$
CON_R_A		≥ 0	*P1*	RelBrd	$connectNum = 1$
			P2	AttBrd	$connectNum = 1$

Sentence level constraint
the spatial-relationship graph must be connected

Fig. 1. Basic entity-relationship (ER) diagrams specification.

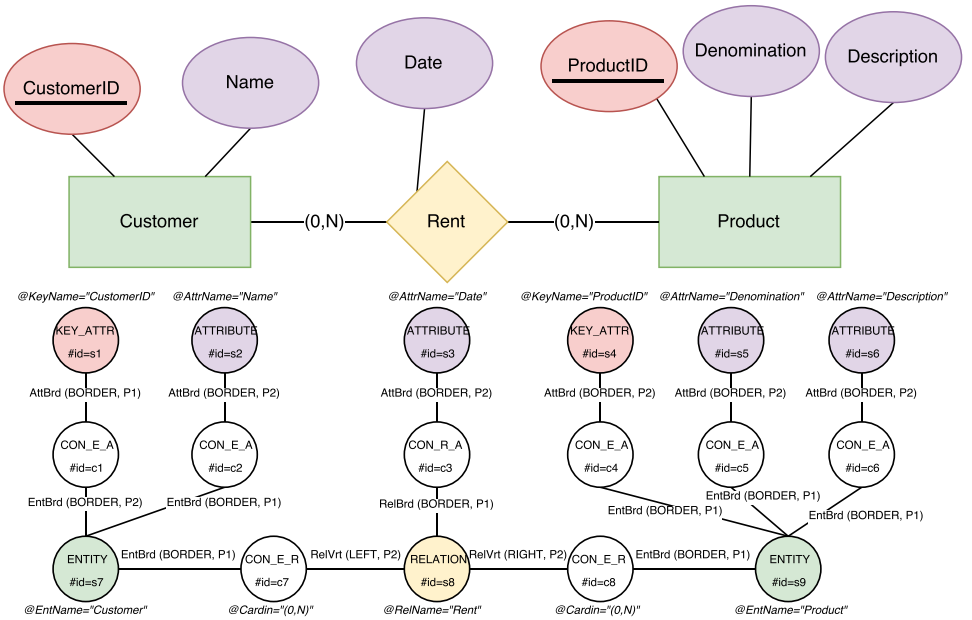


Fig. 2. An ER diagram and its sentence graph.

ENTITY				
Property	**Procedure**	**Params**		**Post-condition**
`$Key : list<string>`	add	`CON_E_A/KEY_ATTR`	`@KeyName`	`size($Key) > 0`
`$Attributes : list<string>`	add	`CON_E_A/ATTRIBUTE`	`@AttrName`	
	addAll	`CON_E_R[@Cardin='([01],1)']/RELATION`	`$Attributes`	

`print(@EntName + " (<u>" + explode(", ", $Key) + "</u>);`
`if(size($Attributes) > 0) print(", " + explode(", ", $Attributes));`
`print(")\n");`				

RELATION				
Property	**Procedure**	**Params**		**Post-condition**
`$Key : list<string>`	addAll	`CON_E_R[@Cardin='([01],N)']/ENTITY`	`$Key`	
`$Attributes : list<string>`	add	`CON_R_A/ATTRIBUTE`	`@AttrName`	
`$Entities : list<string>`	add	`CON_E_R[@Cardin='([01],N)']/ENTITY`	`@EntName`	
`$KeyAttrs : list<string>`	add	`CON_R_A/KEY_ATTR`	`@KeyName`	`size($KeyAttrs) = 0`

`if(size($Entities) == 2) {`
`  print(@RelName + " (<u>" + explode(", ", $Key)) + "</u>");`
`  if(size($Attributes) > 0) print(", " + explode(", ", $Attributes));`
`  print(")\n");`
`}`				

Fig. 3. LCSD specification of ER diagrams built on the syntax definition of Fig. 1.

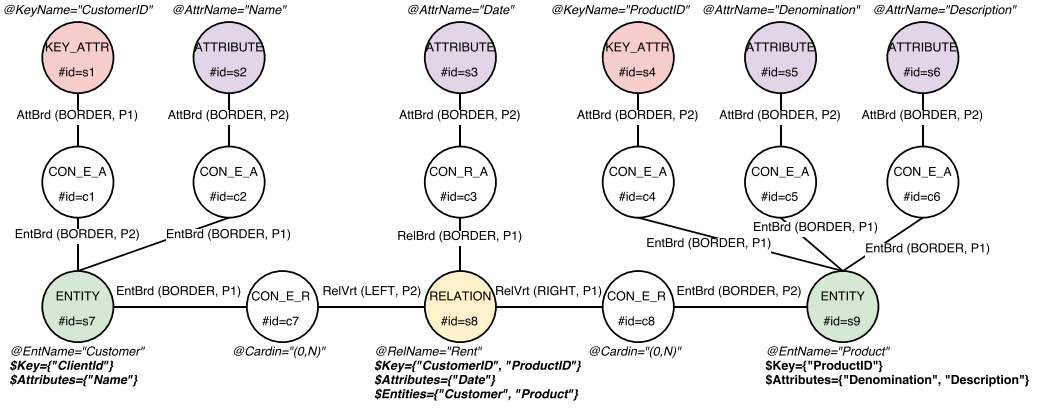


Fig. 4. Sentence graph with properties (in bold) for the ER in Fig. 2.

```
<!ELEMENT language (action? element+ action?)>
<!ATTLIST language
name CDATA #REQUIRED>

<!ELEMENT element (visitpath* (property | action)*)>
<!ATTLIST element
    ref ID #REQUIRED
    priority CDATA #IMPLIED>

<!ELEMENT visitpath (#PCDATA)>
<!ATTLIST visitpath
    type (D|B)>

<!ELEMENT property (function+)>
<!ATTLIST property
    name CDATA #REQUIRED
    type CDATA #REQUIRED
    postcondition CDATA #IMPLIED>

<!ELEMENT function (param+)>
<!ATTLIST function
    name CDATA #REQUIRED>

<!ELEMENT param (#PCDATA)>

<!ELEMENT action (#PCDATA)>
```

Fig. 5. LCSD XML DTD.

```
Input: a sentence graph G and the sets of the semantic rules SR for each
    language element.
Output: SUCCESS, if all the properties have been computed and stored
    into the corresponding nodes in G; ERROR, otherwise
 1: function ApplySemanticRules(G, SR)
 2:     let L be a list of all the graph nodes in G
 3:     remove from L the nodes with no semantic rule in SR
 4:     if the list L is empty then return SUCCESS
 5:     end if
 6:     let x be the first node in L
 7:     calculate all the computable properties of x by using the
            semantic rules in SR defined on the #name of x
 8:     /* actions are considered unnamed properties */
 9:     if no properties of x can be calculated then
10:         if x has no next node in L then
11:             return ERROR
12:         else let x be the next node in L and goto 7
13:         end if
14:     else
15:         if all the properties of x have been calculated then
16:             delete x from L
17:         end if
18:         goto 4
19:     end if
20: end function
```

Fig. 6. The algorithm to compute all the semantic properties of the sentence graph nodes.

3. Local context specification of visual languages

In this section, we recall the local context specification of visual languages as described in [3], add the concept of *text attaching area* and give a detailed description of the sentence graph derived from a visual sentence.

According to the local context specification, a visual language is a set of visual sentences on an alphabet of *symbols* and *connectors* (i.e. *language elements*). Each of them is characterized by the following attributes:

- a unique name;
- a graphical appearance;
- the minimum and/or maximum numbers of admissible occurrences in any sentence of the language;
- one or more attaching areas. Each area is characterized by a *unique name*, its *shape* and *location* on the symbol or connector, a set of *local constraints*, such as the number of possible connections to the area (referred to as *connectNum*), and a *type* used to force legal connections among symbol and connector attaching areas. In fact, a connector area can be attached to a symbol area only if they have the same type;
- a number of symbol level constraints involving more than one attaching area.

Moreover, in order to allow a meaningful visual language translation, in this paper, we introduce the *textual attaching areas*, i.e. attaching areas that are designed to contain text. In this case, the local constraints define the set of admissible values for the contained text. A text value can be constrained to be a string, int, float, boolean or a text described by a regular expression.

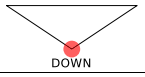
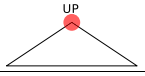
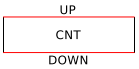
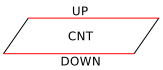
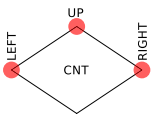
As a further constraint, a local-context based specification may require that the sentence graph (described below) is connected for each sentence of the specified language. This particular constraint will be named *sentence level constraint*.

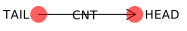
Fig. 1 shows the local context specification of a basic version of the well-known entity-relationship (ER) diagrams. The table consists of two main parts: the first part defines the symbols while the second one defines the connectors. The sentence level constraint is reported in the last row. As an example, the symbol with name ENTITY has a rectangle as a graphical appearance, it must occur at least once in any sentence of the language, it presents a rectangular shaped attaching area with name BORDER overlapping the border of the symbol (with type EntBrd and with local constraint $connectNum \geq 0$), and a textual attaching area EntName located at the center of the rectangle and constrained to be a string.

In addition to checking the sentence correctness, the syntactic analysis outputs a graph, called abstract sentence graph. This contains a node for each symbol/connector and an edge between all the symbol-connector pairs that are related.

Each node is characterized by the following attributes:

- id of the language element;

Symbol name	Graphics	Symbol occurrence	Attaching areas name	type	constraints
BEGIN		1	*DOWN*	exit	$connectNum = 1$
END		1	*UP*	enter	$connectNum \geq 1$
STAT		≥ 0	*UP* *DOWN* *CNT*	enter exit ***text:***Code	$connectNum \geq 1$ $connectNum = 1$ *string*
IO		≥ 0	*UP* *DOWN* *CNT*	enter exit ***text:***Code	$connectNum \geq 1$ $connectNum = 1$ *string*
PRED		≥ 0	*UP* *LEFT* *RIGHT* *CNT*	enter exit exit ***text:***Code	$connectNum \geq 1$ $connectNum = 1$ $connectNum = 1$ *string*

Connector name	Graphics	Connector occurrence	Attaching areas name	type	constraints
ARROW		≥ 0	*HEAD* *TAIL* *CNT*	enter exit ***text:***Rel	$connectNum = 1$ $connectNum = 1$ *(true|false)?*

Sentence level constraint
the spatial-relationship graph must be connected

Fig. 7. Flowchart language specifications.

Priority table	
Name	Priority
BEGIN	-1
END	1
all others	0

BEGIN		
Property	Procedure	Params
$OutId : string	assign	(#attType='exit')::ARROW/* | #id
print("#include <stdio.h>\n\nint main() {\ngoto " + $OutId + ";\n");		

PRED		
Property	Procedure	Params
$GotoTrue : string	assign	(#attType='exit')::ARROW[@Rel='true'] | #id
$GotoFalse : string	assign	(#attType='exit')::ARROW[@Rel='false'] | #id
print(#id + ": if(" + @Cond + ") goto " + $GotoTrue + "; else goto " + $GotoFalse + ";\n");		

IO / STAT		
Property	Procedure	Params
$OutId : string	assign	(#attType='exit')::ARROW/*
print(#id + ": " + @Code + "; goto " + $OutId + ";\n");		

END
print(#id + ": ;\n}");

Fig. 8. Flowchart semantic specification (first version).

- name of the element (e.g.: ENTITY, ATTRIBUTE, etc.);
- a flag indicating whether the node represents a symbol or a connector;
- 0 or more pairs in the format $<$name of the textual attaching area, text value$>$.

An edge representing the relationships between a symbol and a connector has as attributes the type and the names of the two attachment areas of the involved symbol and connector.

Fig. 2 shows an ER diagram and its sentence graph where, for sake of clarity, the ER symbols have been colored. This ER diagram respects all the constraints imposed by the local context specification in Fig. 1. First, it can be noted that the graphic

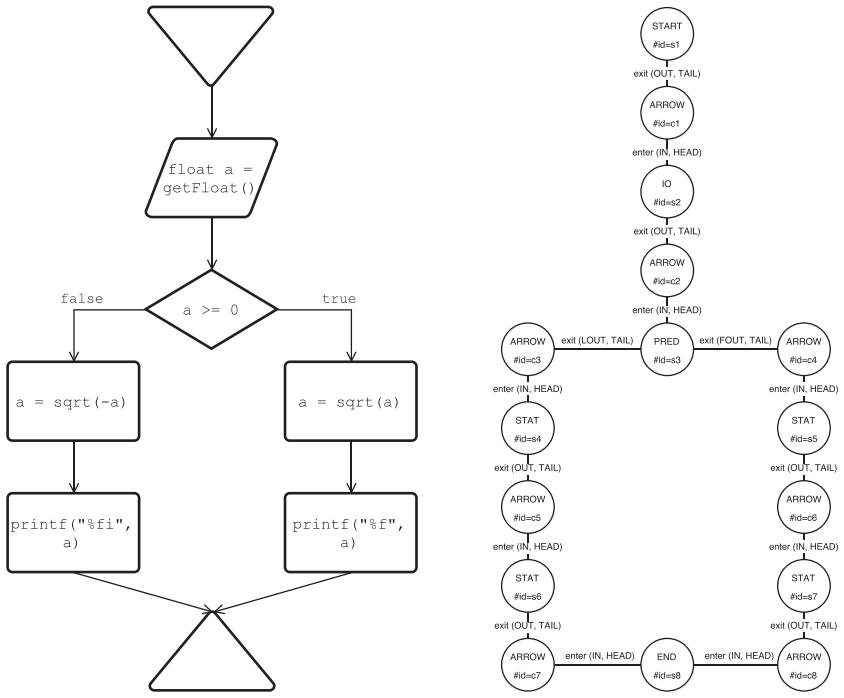


Fig. 9. Flowchart example.

Symbol name	Graphics	Symbol occurrence	Attaching areas name	type	constraints
ROOT	UP CNT DOWN	1	*UP*	enter	$connectNum = 0$
			DOWN	exit	$connectNum \geq 0$
			CNT	***text:***Desc	*string*
NODE	UP CNT DOWN	≥ 0	*UP*	enter	$connectNum = 1$
			DOWN	exit	$connectNum \geq 0$
			CNT	***text:***Desc	*string*

Connector name	Graphics	Connector occurrence	Attaching areas name	type	constraints
EDGE	P1/P2 P2/P1	≥ 0	*P1*	exit	$connectNum = 1$
			P2	enter	$connectNum = 1$

Sentence level constraint
the spatial-relationship graph must be connected

Fig. 10. Tree language specifications.

object *line* corresponds to three different interpretations, i.e., the connectors CON_E_R, CON_E_A, CON_R_A. The name of the corresponding graph node depends on its local context (i.e., the types of its attaching areas). That being said, it is easy to see that each symbol/connector respects all of its local constraints. As an example, the symbol RELATION labelled *Rent* fulfills the first constraint with exactly two total connections on the corners LEFT and RIGHT with two connectors CON_E_R; the only connection on the border with a CON_R_A connector respects the second constraint; the text in the center of the symbol respects the third constraint (i.e., a simple string).

4. Sentence graph navigation with SGPath

This section describes an XPath-like specification for the navigation of the sentence graph called SGPath. While XPath defines path expressions to navigate a hierarchical XML document, SGPath defines path expressions to navigate a graph-like structure.

Path table		
Order	Paths	
ROOT	D	(#attType='exit')::EDGE/NODE
NODE	D	(#attType='exit')::EDGE/NODE

```
ROOT / NODE
print(@Desc + "\n");
```

Fig. 11. Semantic specification for the depth-first visit of the trees specified in Fig. 10.

An SGPath consists of *steps*. Each step is evaluated on a set of sentence graph nodes (named *context*) by considering each element of the set individually. The result of each step is still a set of sentence graph nodes.

Moreover, an SGPath can make use of the following attributes, whose values are mostly taken from the abstract sentence graph:

- `#id`: the id of the language element;
- `#name`: the name of the element (e.g.: ENTITY, ATTRIBUTE, etc.);
- `#isSymbol`: a boolean value indicating whether the node represents a symbol or a connector;
- `#attType`: the type of the attachment areas through which a symbol and a connector are linked;
- `#attConName`: the name of the attachment area (connector side) through which a symbol and a connector are linked;
- `#attSymName`: the name of the attachment area (symbol side) through which a symbol and a connector are linked;
- `#visited`: the number of times that a node/edge has been visited so far during the current path navigation;
- an attribute for each textual attaching area that contains its text (in the form `@textName`);
- user-defined attributes in the form `$userDefName`. The values of these attributes are not read from the sentence graph but must be calculated (as described in Section 5). When an SGPath expression refers to such an attribute that has not yet been calculated then the SGPath evaluation is interrupted and an appropriate message is returned. To distinguish these particular attributes, in the following, we will refer to them as *properties*;
- `#status`: the status of the calculation of the *properties* of a node as a whole. It can be COMPLETE or INCOMPLETE.

The syntax of an SGPath is therefore of the form:

```
sgpath_step_1/sgpath_step_2/ … /sgpath_step_n
```

and each `sgpath_step_i` is of the form:

```
axis(edge-filter)::node-test[predicate]
```

where:

- `axis(edge-filter)::` is optional and indicates how to continue the navigation starting from the current node. Unlike XPath where the `axis` allows to select nodes at different distances from the current node, SGPath is limited to select the current node (keyword `self`) or the nodes immediately adjacent, i.e., connected directly with a single edge to the current node (keyword `near`). If no keyword is specified, `near` is used by default.
  The field `edge-filter` allows the specification of boolean constraints on the graph edges that it is possible to use to reach the neighboring nodes and, more specifically, on their attributes (`#attType`, `#attConName`, `#attSymName` and `#visited`). If this field is absent then all the edges are taken into account.
- `node-test` specifies the node to be reached. It can be an *element name* or * to indicate any node.
- `[predicate]` is an optional boolean expression that sets conditions on the existence of nodes with specific attribute values. Unless otherwise specified by a path the attributes of the current node (*self*) are checked.

Moreover, it is possible to combine SGPaths by using the `union`, `except` or `intersect` operators on sets of sentence graph nodes.

In order to describe some examples of paths we consider the graph in Fig. 2. For space reasons some attributes are shown in synthetic form. By convention, the nodes with colored background have `#isSymbol=true`, while the ones with blank background have `#isSymbol=false`. In particular, the node with `#id=s1` has `#name=KEY_ATTR`, `#isSymbol=true` and `@KeyName='CustomerID'`, while the leaving edge has `#attType=AttBrd`, `#attSymName=BORDER` and `#attConName=P1`.

In the following SGPath examples, in order to reduce verbosity, *node NNN* will denote a node with `#name=NNN`, while *node sn* will denote the node with `#id=sn`, and *{sx, sy}* will denote the set containing the nodes with `#id=sx` and `#id=sy`.

- `CON_E_A/ATTRIBUTE` denotes all the `ATTRIBUTE` nodes reachable by navigating the graph from the current node and necessarily going in sequence through an edge, a `CON_E_A` node, and an edge. In this example predicates and axis are not present.
  If the current node is s7 the path denotes {s2}; if the current node is s9 the path denotes {s5, s6}.
- `CON_E_A/ATTRIBUTE[#visited=1]` adds to the previous path a predicate that requires that the resulting `ATTRIBUTE` nodes be considered only if they have already been visited once along the path graph navigation (including the start node). If a predicate on `#visited` is not present then `[#visited=0]` is assumed, i.e., each node in the path can be considered only if not visited previously in the path graph navigation.
  If the current node is s2 the path denotes {s2}. If the predicate is omitted the path denotes the empty set.

Visit table			
Order	**Priority**	**Paths**	
BEGIN	1	D	ARROW
ARROW	1	D	(#attName='HEAD')::*
STAT	1	D	(#attType='exit')::ARROW
IO	1	D	(#attType='exit')::ARROW
PRED	1	D	(#attType='exit')::ARROW[@Rel='true']
		D	(#attType='exit')::ARROW[@Rel='false']
END	2		

BEGIN
print("#include <stdio.h>\n\nint main() {\n");

PRED			
Property	**Procedure**	**Params**	
$GotoTrue : string	assign	(#attType='exit')::ARROW[@Rel='true']/*	#id
$GotoFalse : string	assign	(#attType='exit')::ARROW[@Rel='false']/*	#id
$Count : int	size	(#attName='UP')::*	
$PrePred : boolean	exist	(#attName='UP')::ARROW/PRED	
if($Count > 1 || $PrePred) print(#id + ": "); print("if(" + @Cond + ") goto " + $GotoTrue + "; else goto " + $GotoFalse + ";\n");			

ARROW			
Property	**Procedure**	**Params**	
$Goto : string	assign	(#attName='HEAD')::*	#id
$Printed : int	assign	(#attName='HEAD')::*	#status
$PrePred : boolean	exist	(#attName='TAIL')::PRED	
$End : boolean	exist	(#attName='HEAD')::END	
if(!$PrePred && ($Printed == COMPLETE || $End)) print("goto " + $Goto + ";\n");			

IO / STAT		
Property	**Procedure**	**Params**
$Count : int	size	(#attName='UP')::*
$PrePred : boolean	exist	(#attName='UP')::ARROW/PRED
if($Count > 1 || $PrePred) print(#id + ": "); print(@Code + ";\n")		

END
print(#id + ": ;\n}");

Fig. 12. Flowchart semantic specification (second version).

- `*/*/ATTRIBUTE` uses the wildcard `*`.
  If the current node is c7 this path denotes {s2, s3}. The first `*` allows us to reach both the `ENTITY` node s7 and the `RELATION` node s8 while the second one allows us to reach the `CON_E_A` nodes c1 and c2, and the `CON_E_R` node c3.
- `CON_E_R[@Cardin='([01],1)']/RELATION` denotes all the `RELATION` nodes reachable from the current node by sequentially using an edge, a `CON_E_R` node with the `@Cardin` text attribute equal to `(0,1)` or `(1,1)` (the string `'([01],1)'` is evaluated as a regular expression) and an edge.
  If the current node is s7 this path denotes an empty set. Without the predicate `[@Cardin='([01],1)']` the set would be {s8}.
- `CON_E_R/ENTITY[size($Attributes)=2]` denotes all the `ENTITY` nodes which have the property `$Attributes` equals to a set of two elements and that can be reached from the current node by sequentially using an edge, a `CON_E_R` node and an edge.
  By looking at Fig. 4 in Section 5, if the current node is s8 this path denotes {s9}; without the predicate, the set would be {s7, s9}.
- `CON_E_R/ENTITY[CON_E_A/KEY_ATTR[@KeyName='CustomerID']]` denotes the `CON_E_R/ENTITY` nodes with the requirement that there exists at least one `KEY_ATTR` node with `@KeyName` equals to *"CustomerID"* reachable by navigating the graph from the `CON_E_R/ENTITY` nodes and necessarily going in sequence through an edge, a `CON_E_A` node, and an edge.
  If the current node is s8 this path denotes {s7}; without the predicate, the set would be {s7, s9}.
- `(#attType='RelBrd')::*` denotes all the nodes reachable from the current node by using an edge with attribute `#attType` equal to "RelBrd".
  If the current node is s8 this path denotes {c3}; without the predicate, the set would contain all the nodes adjacent to s8, i.e. {c3, c7, c8}. If `(#attType='RelBrd')` is substituted with `(#attSymName='LEFT')` the set would instead be {c7}.

- `self()::ENTITY` denotes the current node with symbol name `ENTITY`. If the current node is s7 this path denotes {s7}; if the current node is s8 this path denotes the empty set.

5. Local context-based semantic definition and analysis

In this section, we present a new way to define a semantic translation for a visual language. To do so, we describe the Local Context-based Semantic Definition (LCSD) and its evaluation algorithm. The LCSD consists of a sequence of *semantic rules* for each element of the language. Each rule either calculates a *property* or executes an *action*. The properties are calculated through *procedures* making use of SGPath expressions and possibly validated through a *post-condition*. An action depends on properties and attributes.

Through the post-conditions, an LCSD may better refine the syntactic structure of the language sentences; through the actions, it provides a translation of the sentences. Finally, the calculated properties can also be used to produce an annotated sentence graph to be exploited in possible successive synthesis phases.

Fig. 3 shows an LCSD in tabular format for a basic version of the ER diagrams referring to the Local Context specification of Fig. 1. The two tables provide the semantic rules for the symbols ENTITY and RELATION, respectively. The properties of each symbol are listed in the first column with their names (prefixed with $) and types. In the subsequent columns, each property is followed by the name of the procedures used to give a value to it together with their parameters, and possible post-conditions. The last row of each table shows an action for the symbol.

Note that, by definition of the LCSD, the order of the table rows is relevant and a property is calculated through one or more procedures (see the property `$Attributes` in table ENTITY).

In general, the procedures set/change the value of a property and take as parameters a set of sentence graph nodes selected through an SGPath and the attributes/properties to be read from the nodes. In Fig. 3, the procedure `add` in the table ENTITY, corresponding to the property `$Key`, derives the set of key attribute names of the considered Entity, from its parameters. First, it selects the nodes `KEY_ATTR` connected through the `CON_E_ATTR` nodes to the current ENTITY node and then, for each of them, it adds their `@KeyName` to the list of strings `$Key`. The procedure `add` corresponding to the property `$Attributes` adds to it the names of the Attributes connected to the Entity, while the `addAll` function in the next row adds to it all the Attributes of the Relations connected to the Entity with cardinality 0,1 and 1,1 (the difference between `add` and `addAll` is that the second argument is a singleton in the first case and a list in the second one).

A post-condition is used to constrain the possible values of a property. In Fig. 3, the constraint `size($Key) > 0` indicates that an Entity must have at least one key attribute, while `size($KeyAttrs) = 0` indicates that a Relation cannot have key attributes. If the constraints are not respected the sentence will not be accepted.

The action rules are generally used to produce an output (e.g. a textual translation of the visual sentence). They are written in any programming language depending on the implementation. They usually reference the properties and attributes of the language element. In the ENTITY table, the action rule prints the entity data in the output relational model format.

In our example, the LCSD definition in Fig. 3 applied to the ER diagram in Fig. 2 produces the annotated sentence graph in Fig. 4 and the following text (the detailed evaluation algorithm is shown in the next section):

```
Customer (CustomerID, Name)
Product (ProductID, Denomination, Description)
Rent (CustomerID, ProductID, Date)
```

Formally, an LCSD is described in XML format according to the DTD in Fig. 5. This completely describes the tabular format with the addition of two optional actions, at the beginning and at the end of the specification, to allocate and deallocate, respectively, needed resources (e.g. files, etc,). Appendix A shows the ER LCSD specification in XML format.

5.1. The LCSD evaluation algorithm

The algorithm in Fig. 6 is used on the sentence graph in order to calculate the value of the properties for each node (it is to remember that each element in the sentence corresponds to a node in the graph).

Given that, as already mentioned, the rules for the calculation of the properties may refer to other properties (that may have not yet been calculated), the algorithm works incrementally by calculating at each step the properties that can be calculated. In detail, the algorithm behaves as described below.

The algorithm takes as input a sentence graph G (as described in Section 3) and the sets of the semantic rules RS for each language element.

As a first step, the algorithm calculates the list L of the graph nodes and then proceeds by trying to calculate all the properties of each node in L. For sake of simplicity, here the actions are considered as unnamed properties. In particular, the algorithm tries to calculate all the properties of the first node in L using the semantic rules in SR, in case that it is not possible to calculate any properties (due to the fact that all semantic rules refer to properties that have not yet been calculated) the algorithm moves to the next node in L and tries the procedure on it and so on (if there are no more nodes in L the algorithm terminates with an error status). If it was possible to calculate at least one property of the node, the

algorithm restarts as described above from the first remaining node in L. In the case that all the properties of the node have already been calculated, the node must be removed from L before restarting.

The algorithm terminates for any input, this happens because if after analyzing all the nodes of L no new property has been calculated, the algorithm terminates with an error status. If conversely at least one new property is calculated, the algorithm converges to the solution, given that the number of properties still to be calculated has been decremented.

It is noteworthy that the error status normally occurs when the sentence does not respect the constraints defined by the set of semantic rules. In this way, the semantic rules can be used not only to calculate the properties and produce an output, but also to specify the language sentences more precisely.

The error status can of course also occur in the case that the language specification is incorrect. It is possible to imagine the creation of a dependency graph based on the specification of the semantic rules in order to verify that the specification allows for at least one sentence for which the algorithm does not return the error status (although in practice this is not of great utility for the user). In general, as said before, it is not necessary to ensure that the dependencies are satisfiable for all the possible sentences.

As an example, let us consider the algorithm execution on the ER in Fig. 2. Since there are semantic rules only for `ENTITY` and `RELATION` nodes, after the execution of line 3, the list L will be composed of the nodes s7, s8, and s9 (this is an arbitrary order).

As a first step, the algorithm will consider *x=s7* and will try to calculate all of its properties. The properties `$Key` and `$Attributes` will be calculated (`$Key={'CustomerID'}`, `$Attributes={'Name'}`) and the semantic action will be performed (printing "`Customer (CustomerID, Name)`"). s7 will then be removed from L.

The algorithm will then start again from line 3 (L is not empty), and consider *x=s8* and will try to calculate all of its properties. It will not be possible to calculate its property `$Key` because the property `$Key` of node s9, on which it depends, has not been calculated yet. As a consequence, also its semantic action cannot be calculated. On the other hand, its properties `$Attributes` and `$Entities` will be calculated (`$Attributes={'Date'}`, `$Entities={'Customer', 'Product'}`).

The algorithm will then start again from line 3 (L is not empty), again selecting *x=s8*, but since it is not possible to calculate any additional properties, it will select *x=s9* in line 11 and will try to calculate all of its properties. The properties `$Key` and `$Attributes` will be calculated (`$Key={'ProductID'}`, `$Attributes={'Denomination', 'Description'}`) and the semantic action will be performed (printing "`Product (ProductID, Denomination, Description)`"). s9 will then be removed from L.

The algorithm will then restart, once again taking into consideration *x=s8*, this time it will be possible to calculate the remaining property (`$Key={'CustomerID', 'ProductID'}`) and to perform the semantic action (printing "Rent (CustomerID, ProductID, Date)"). s9 will then be removed from L and the algorithm will end returning SUCCESS. All the calculated properties will have been stored on the sentence graph as shown in Fig. 4.

5.1.1. Time complexity considerations

Let n be the number of nodes in G and m the maximum number of semantic rules for each language element in SR. The bulk of the algorithm work is given by the steps in which it tries to evaluate the properties (including semantic actions) unsuccessfully. If P_iL_j indicates the ith property of the jth element of L, the worst case occurs when the algorithm has to evaluate the properties in the order given by the following sequence:

$< P_iL_j, P_iL_{j-1} >$ for $i = m, m-1, \ldots, 1$ and $j = n, n-2, n-4, \ldots, 2$

By iterating on i and j, the explicit sequence can be split in $n/2$ rows:

$$P_mL_n,\ P_mL_{n-1},\ P_{m-1}L_n,\ P_{m-1}L_{n-1},\ \ldots,\ P_1L_n,\ P_1L_{n-1}, \quad (1)$$

$$P_mL_{n-2},\ P_mL_{n-3},\ P_{m-1}L_{n-2},\ P_{m-1}L_{n-3},\ \ldots,\ P_1L_{n-2},\ P_1L_{n-3}, \quad (2)$$

$$\ldots, \quad (\cdots)$$

$$P_mL_2,\ P_mL_1,\ P_{m-1}L_2,\ P_{m-1}L_1,\ \ldots,\ P_1L_2,\ P_1L_1. \quad (n/2)$$

The number of properties evaluations needed to calculate the property P_iL_j is given by the number of the properties of the elements preceding L_j, i.e., the m properties of each of the $j-1$ preceding elements, plus $i-1$ evaluations of the properties preceding P_i and the evaluation of P_i. The final number is then given by $i + (j-1) \times m$. Note that, due to the ordering above, all the properties of the elements preceding L_j are yet to be calculated.

As a consequence, the number of properties/semantic actions evaluations for row (1) is:

$$\begin{aligned}
&(m + (n-1) \times m) && + && (m + (n-2) \times m) \\
&+((m-1) + (n-1) \times m) && + && ((m-1) + (n-2) \times m \\
&+\cdots && + && \cdots \\
&+(1 + (n-1) \times m) && + && (1 + (n-2) \times m)) \\
&= \frac{m(m+1)}{2} + (n-1) \times m^2 && + && \frac{m(m+1)}{2} + (n-2) \times m^2 \\
&= m(m+1) + (n-1) \times m^2 + (n-2) \times m^2
\end{aligned}$$

For row (2): $m(m+1) + (n-3) \times m^2 + (n-4) \times m^2$.

...

For row (n/2): $m(m+1) + (1) \times m^2 + (0) \times m^2$.

In total, the number of evaluations will be $(n/2) \times m(m+1) + \frac{n(n-1)}{2} \times m^2$.

We need now to calculate the time complexity of the evaluation of the properties and the semantic actions. As regards the properties, the principal work is the SGPath evaluation. If we assume an SGPath with s steps of the form `*[#visited>=0]/` and a connected graph G with n nodes, the complexity of the SGPath evaluation will be $O(n + (s-1)n^2)$ since each step will evaluate n nodes for each of the previous steps evaluated nodes.

If instead the steps follow the default constraint `[#visited=0]` each node can be visited only one time, and so the complexity will be $O((n/s)^s)$ (it can be shown that, given a set of numbers with a fixed sum (n in our case), the maximum product is achieved when the numbers are equal).

Moreover, if each graph node has at most e incident edges, the complexity will become $O(e^s)$. Since both e and s are small in a typical visual language, in this case, the complexity can be assumed to be $O(1)$.

As regards semantic actions, the computation complexity is unbounded in the general case, since the source code is an input of the language designer. For the typical visual language, however, they usually consist of simple print operations, so we can also assume a complexity of $O(1)$.

Under these assumptions, the algorithm time complexity in the worst case is $O(n^2m^2)$. In the typical case of $m^2 \ll n^2$ the complexity can be assumed to be $O(n^2)$.

5.2. Graph nodes processing order

Given a sentence graph, the LCSD algorithm in Fig. 6 calculates the properties and actions of the nodes in a data flow-driven way without any order constraint on its execution flow.

For handling more complex semantic evaluations and to improve the execution efficiency of the algorithm, we modify line 2 of the algorithm in Fig. 6 by setting a partial order on the sentence graph nodes whose properties and actions must be calculated.

5.2.1. Ordering with priority values

A simple way to specify a partial order is to declare a priority value for each of the language elements (symbols/connectors), as shown in the *priority table* at the top of Fig. 8. This table specifies a priority per each element of the flowchart language of Fig. 7, in particular −1 for the symbol `BEGIN`, 1 for the symbol `END`, and 0 for all the other symbols/connectors. In this way, the `BEGIN` nodes will appear at the beginning of the list (the priorities are considered in ascending order), the `END` nodes at the end of the list, and the remaining nodes in arbitrary order. For example, the list for the flowchart in Fig. 9 will be {s1, *other symbols/connectors in any order*, s8}.

By knowing that the symbols/connectors will be processed by the algorithm in this order, it is simple to produce the semantic specification that translates a flowchart to a C code (with goto's), as shown in Fig. 8.

In particular, the `assign` procedure in the table BEGIN for the property `$OutId` derives from its parameters the `#id` of the single node corresponding to the symbol connected to the `BEGIN` symbol through an `ARROW`. By definition, unlike the `add` procedure seen previously, the `assign` procedure works correctly only if the path given as argument returns exactly one single node, otherwise, it throws a syntax error exception. Therefore this semantic adds a further constraint to the language (in a similar manner to a postcondition `size($prop)=1` in the case of a list), although in this particular case this situation cannot occur since the local constraints already ensure that the path evaluates to a single node. The BEGIN semantic action prints the program header and a goto to the first instruction.

The table PRED prints an if-else statement with goto's and thanks to the `assign` procedure ensures that the symbol is connected exactly to one arrow with text "true" and to one arrow with text "false" (in this case, these constraints are not guaranteed by the local constraints). The semantic for the symbols Io and Stat, shown in the IO / STAT table, is very similar and prints a label, the corresponding instruction and a goto to the next instruction (assigned to the property $OutId). In the table END the semantic action prints the program end goto label and closes the program with a `}`. One of the possible outputs for the diagram in Fig. 9 is:

```
#include <stdio.h>

int main() {
goto s2;
s4: a = sqrt(-a); goto s6;
s2: float a = getFloat(); goto s3;
s7: printf("%f", a); goto s8;
s5: a = sqrt(a); goto s7;
s6: printf("%fi", a); goto s8;
s3: if(a >= 0) goto s5; else goto s4;
s8: ;
}
```

Although this simple way of defining the order can be useful in some cases, it does not allow to define more complex visits. For example, in the case of the tree language shown in Fig. 10, a different priority for ROOT and NODE will not make possible to specify a depth-first or breadth-first order visit.

5.2.2. Ordering with SGPaths

A more expressive and distributed way to define an order is shown in the Path table in Fig. 11. This table defines a total order on the language elements and, for each of them, a list of SGPaths to indicate how to navigate from an element to the next one. Moreover, each SGPath is annotated with a flag D or B indicating whether its resulting nodes must be processed in a depth-first or breadth-first way, respectively.

Informally, the procedure used to calculate the order is as follows: (1) select a not yet processed node (giving precedence to the symbols/connectors that appear first in the *Order* column) and add it to the output node list; (2) evaluate its corresponding path in the *Paths* column; (3) for each node *n* resulting from this evaluation and not in the output list, add *n* to the list and apply its corresponding path recursively as in step (2) (the way the nodes are inserted in the list depends on the value D or B of the flag). The procedure ends if all the nodes have been added to the list. In the case all the recursive steps have been performed and the list still does not contain all the graph nodes the procedure returns to step (1).

It is easy to see that this procedure, when applied to the Path table in Fig. 11, will produce a depth-first order visit. This, combined with the semantic actions of ROOT and NODE shown on the right side of Fig. 11, will produce the printing of all the tree nodes in depth-first order.

5.2.3. Ordering algorithm

To allow even more expressiveness, these two specification modes have been combined. As a result, the node order is specified by defining a total order on the corresponding language elements and, for each of them, a priority value and a list of SGPaths to indicate how to navigate from an element to the next one. An example is given by the Visit table at the top of the flowchart semantic specification in Fig. 12.

More formally, the algorithm in Fig. 6 is modified by defining L in line 2 as the result of the invocation of the function ApplyVisitTable shown in Fig. 13. This function implements the combined mode described above and returns a list L of all the graph nodes ordered according to a given visit table.

At the beginning, the function ApplyVisitTable selects the first node *rem* in N (line 13) by using the order given by the list *O* (note that N is the complement of L in G). The following call to the procedure FollowPath selects a list of *nodes*, starting from *rem*, by evaluating sequences of paths in *VTPATHS*. These *nodes* are then added to *L* (line 25) and removed from N (line 26).

If *L* does not contain all the nodes in the graph G then a new iteration will select one of the nodes not in *L* using the order given by the list *O*. When L contains all the nodes of the graph, L is reordered such that the nodes with a name that corresponds to a lower priority value in *PRIORITY* precede in L elements with a name having a higher priority. Elements with the same priority maintain their existing relative order in L.

The procedure FollowPath selects the first list of *nodes*, starting from a given node, by evaluating in order the paths in *VTPATHS* corresponding to that node. It then proceeds in a recursive manner on the selected nodes by applying the paths until no additional node can be selected. It is important to note that nodes not in N are no longer taken into account.

As an example, we apply the function ApplyVisitTable to the case of a flowchart language. The local context specification of our flowchart language is shown in Fig. 7, while its LCSD including its visit table is shown in Fig. 12.

Taking into account the diagram in Fig. 9, the algorithm proceeds as follows: since the first element in the *O* list is BEGIN, a BEGIN node must be considered. Since s1 is the only BEGIN node, it will be the first element added to L. During the FollowPath execution, the path * corresponding to BEGIN is evaluated (line 35), causing c1 to be added to *nodes* (line 39). The evaluation of the path `(#attName='HEAD')::*` corresponding to ARROW will lead to the inclusion of s2 in *nodes*. By continuing in a similar manner c2 and s3 will be added, after which the first path relative to PRED will be applied, leading to the adding of c3 and subsequently of s4, c5, s6, c7, and s8. Given that there are no paths defined for END, after the s8 addition, the second path of PRED will be followed therefore comporting the addition of c4, s5, c6, s7 and c8 (s8 is not added). The outputs of FollowPath is *nodes* = {s1, c1, s2, c2, s3, c3, s4, c5, s6, c7, s8, c4, s5, c6, s7, c8 }. All the elements of *nodes* are then added to L and since $N = \emptyset$ the list is rearranged according to the priorities in *PRIORITY* thus moving the single END node (s8) to the end of L (L={s1, c1, s2, c2, s3, c3, s4, c5, s6, c7, c4, s5, c6, s7, c8, s8}).

The flowchart specification in Fig. 12 uses two procedures not yet seen in the previous examples: `size`, that returns the number of the node defined by the input path and `exist`, that returns true if the input path evaluates to at least a node (or false otherwise).

This specification differs from the simpler one in Fig. 8 because it prints the program instruction in depth-first order and, in doing so, omits some gotos and relative labels. In particular, the semantic action of ARROW prints a goto if its tail is not connected to a PRED symbol (it already prints its gotos) and the symbol connected to its head has been completely processed or it is an END. In the PRED, IO, STAT symbol definitions instead, the goto label is printed only if the symbol has more than one incoming arrow or a single arrow that come from a PRED symbol.

```
Input: a sentence graph G and a visit table PT on the symbols/connectors.
Output: the ordered list L of all the graph nodes in G based on PT
 1: function ApplyVisitTable(G, PT)
 2:     # PT.O is an ordered list that contains some (or all) of the connector/symbol
          names (from the local context specification)
 3:     # PT.PRIORITY is a map that maps a connector/symbol name to an integer
 4:     # PT.PATHS is a map that maps a connector/symbol name to a list of flagged
   SGPaths
 5:     let N be a sorted set of the nodes in G sorted by name, using the order in PT.O
 6:     # nodes with the same name are placed in an arbitrary order, and the nodes with
   a name that is not in PT.O are placed at the end
 7:     let L be an empty list
 8:     while N ≠ ∅ do
 9:         let rem be the first element of N
10:         remove rem from N
11:         add rem at the end of L
12:         let nodes be an empty list
13:         FollowPath(rem, G, N, PT.PATHS, nodes)
14:         add all the elements of nodes at the end of L
15:         remove all the elements of nodes from N
16:     end while
17:     order L such that the nodes with a name that corresponds to a lower value in
        PT.PRIORITY precede elements with a name with a higher priority (elements
        with the same priority maintain their existing relative order)
18:     return L
19: end function

Input: a starting node nnode, a sentence graph G, a node set N, a map PTPATHS that
    maps a connector/symbol name to a list of flagged SGPaths, a visited node list nodes.
Output: the visited node list nodes with all the nodes in both G and N reachable from
    nnode using the SGPaths in PTPATHS.
20: procedure FollowPath(nnode, G, N, PTPATHS, nodes)
21:     let nname be the name associated to the node nnode
22:     let npaths be the list of flagged paths in PTPATHS associated to the name nname
23:     for all npath in npaths do
24:         let nds be the set of nodes in G reached by following the path npath starting
              from the node nnode
25:         remove all the elements of nodes from nds
26:         remove all the elements not in N from nds
27:         if npath.flag = B then
28:             add all the elements of nds at the end of nodes
29:         end if
30:         for all n in nds do
31:             if npath.flag = D then
32:                 add n at the end of nodes if not already present
33:             end if
34:             FollowPath(n, nodes, G, N, PTPATHS)
35:         end for
36:     end for
37: end procedure
```

Fig. 13. The function ApplyVisitTable.

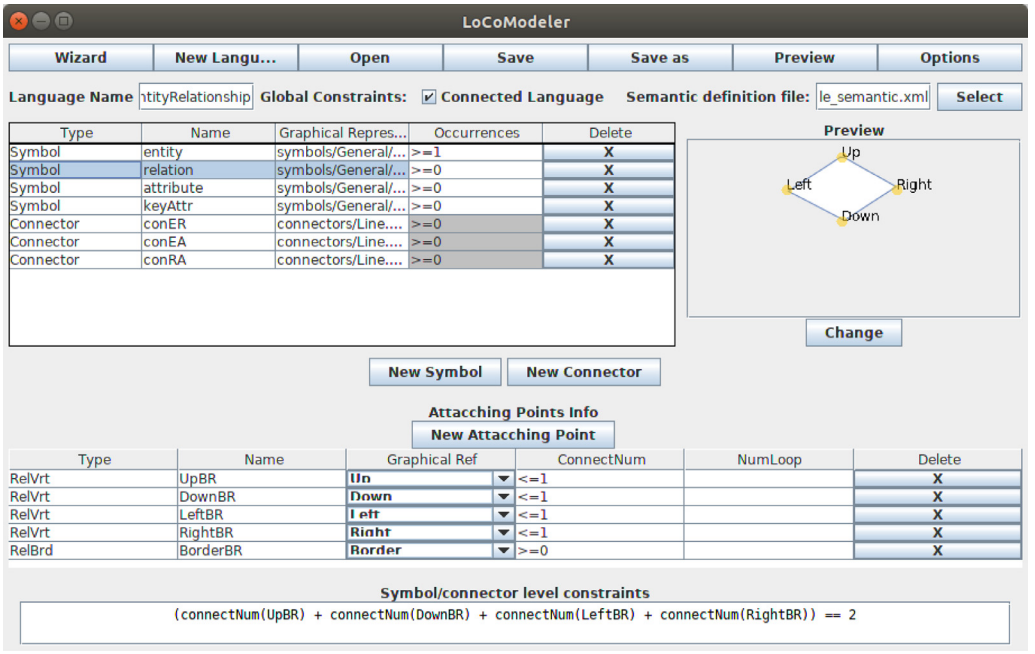


Fig. 14. LoCoModeler: the entity-relationship diagram definition with the semantic definition file loaded.

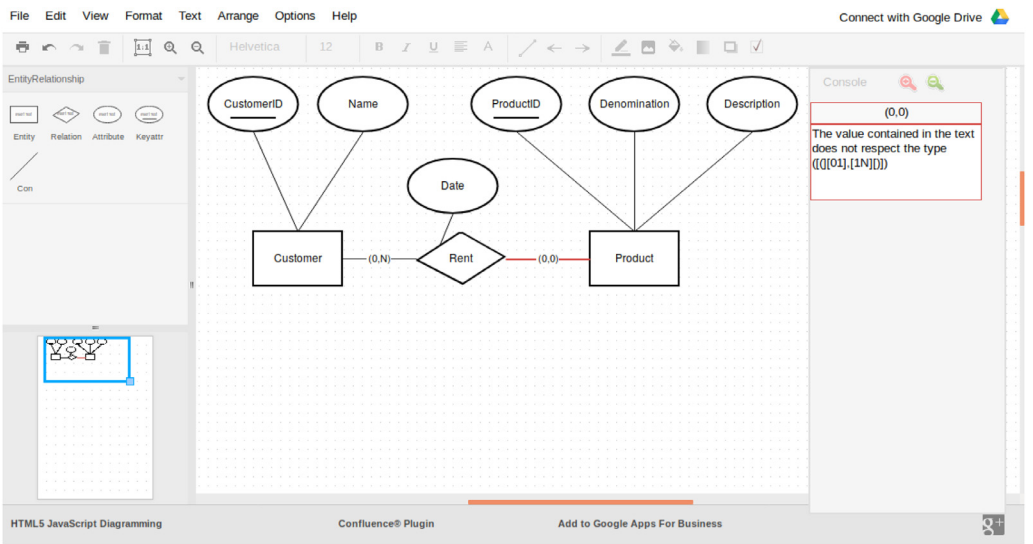


Fig. 15. TiVe: a failed entity-relationship diagram verification.

The output for the diagram in Fig. 9 is:

```
#include <stdio.h>

int main() {
float a = getFloat();
if(a >= 0) goto s5; else goto s4;
s5: a = sqrt(a);
printf("%f", a);
goto s8;
s4: a = sqrt(-a);
printf("%fi", a);
goto s8;
s8: ;
}
```

5.2.4. Further time complexity considerations

First, we analyze the complexity of the FollowPath procedure. Be p the maximum number of paths for each symbol/-connector name in *PTPATHS*, the *for* instruction in line 23 will be repeated at maximum p times. The bulk of the work in the *for*, before the recursive call, is the SGPath evaluation, since, with the proper data structure implementation, the removal of elements in *nds* in lines 25 and 26 can be assumed not to increase its complexity. Since the recursion stop condition occurs when *nds* is empty (line 30), the number of SGPath evaluations is $O(p \times m)$, where m is the total number of elements added to *nodes*.

Regarding the ApplyVisitTable function, let n be the number of nodes in G. Building the sorted set N involves a cost of $O(n \log n)$.

In the body of the *while* (line 8), the reading and removal of the first element of N has a cost of $O(\log |N|)$ (we assume that N is a red-black tree), while adding it at the end of L has a cost of $O(1)$.

Let m be the number of elements in *nodes* after the call of FollowPath, if we assume that s is the cost of an SGPath evaluation, the cost of the FollowPath call is at most $O(p \times m \times s)$.

The addition of the m elements of *nodes* at the end of L has a cost of $O(m)$, while their removal from N has a cost of $O(m \log |N|)$.

From this, we can deduce that the cost of the *while* body is $O(p \times m \times \max(s, \log |N|))$, where m is also the number of elements that have been removed from N during its execution.

Since the *while* loop will be repeated until N is empty, the complexity is $O(p \times n \times \max(s, \log n))$. In the typical visit table case, p will be a very small number, and the cost of an SGPath evaluation will be at most $O(n)$ (see Section 5.1 for some reasoning), so we can assume that the complexity of ApplyVisitTable is $O(n^2)$.

Under these assumptions, the complexity of the algorithm in Fig. 6 will not change with the addition of the call to ApplyVisitTable.

6. The tool LoCoMoTiVE

The local context-based semantic analysis is currently implemented by extending the tool LoCoMoTiVE (Local Context-based Modeling of 2D Visual language Environments) already presented in [23]. This tool is composed of two modules:

- LoCoModeler: the local context-based specification editor that allows designers to create and edit visual language specifications based on local context as given in Section 3, and to produce the TiVE editor.
- TiVe: a web visual language environment for drawing and checking the correctness of the visual sentences. Its implementation is based on Draw.io (https://www.draw.io).

The tool has been extended by allowing the LoCoModeler to load a Local Context Sematic Definition in XML format and produces a TiVE able to show the result of the semantic translation or possible error messages.

The screenshot in Fig. 14 shows the LoCoModeler. It allows to visually input a Local Context Specification and to load a Local Context Semantic Definition in XML format as the one shown in Appendix A.

Three screenshots of TiVE are shown in Figs. 15–17. Fig. 15 shows an incorrect ER diagram: the line connecting Rent and Product (shown in red) has a text not respecting the pattern as indicated in the specification. In this case, the output panel on the right presents an error message. On the other hand, Figs. 16 and 17 show a correct ER and a flowchart diagrams, respectively. For both, TiVE returns their translation as specified in the semantic actions.

An implementation of the tool can be downloaded at the address http://cluelab.di.unisa.it/locomotive.

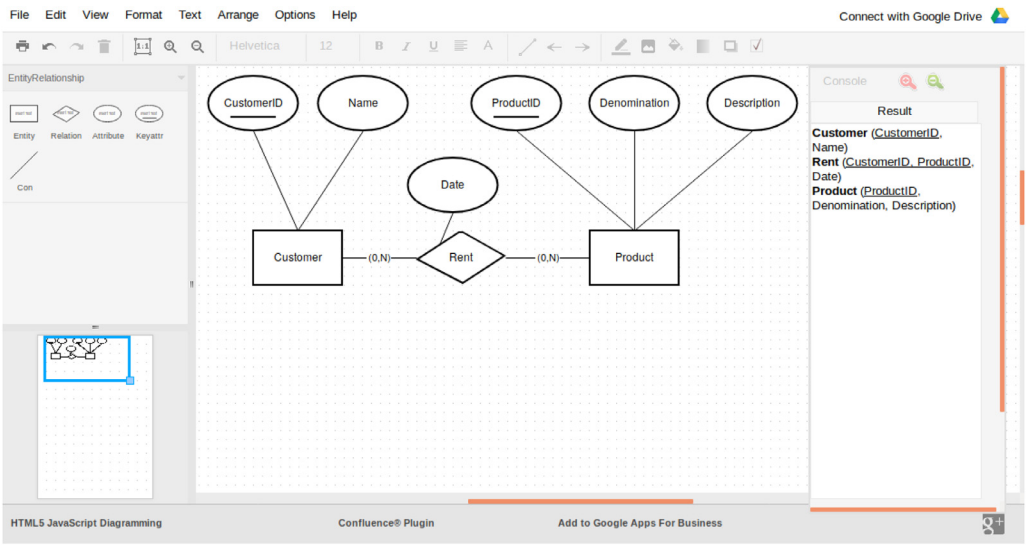


Fig. 16. TiVe: a successful entity-relationship diagram verification with the resulting semantic translation shown on the right.

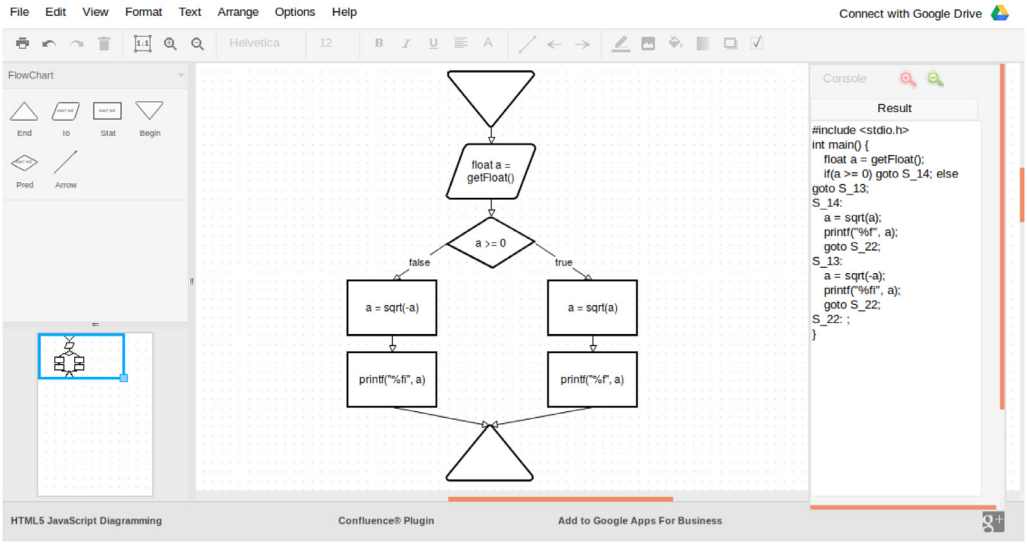


Fig. 17. TiVe: a successful flowchart diagram verification with the resulting semantic translation shown on the right.

7. Conclusions

In this paper, we have presented a new technique for a local context based semantic translation of a visual language. The technique uses XPath-like expressions together with a data flow model of execution. As for the case of local syntax checks, attributes and rules to calculate them are defined for each element of the language. For a given element in the abstract sentence graph, the SGPath expressions are used to gather values from its neighbors in order to allow the rules to calculate its semantic attributes. The new methodology has been implemented in the tool LoCoMoTiVe and been tested on visual languages such as entity-relationship diagrams, flowcharts, trees.

As future work, we are extending the LoCoMoTiVe tool with a visual interactive interface in order to provide an easy way to compose local context specifications both for the syntax and the semantics of visual languages as proposed in this paper. The new module will automatically compose the XML specifications allowing a language designer to compose/visualize/reason about his/her specifications and test them in the same tool. Furthermore, we will apply our methodology to more and more visual languages.

Acknowledgment

The authors wish to thank Fabio Favale, Alfonso Ferraioli and Luca Laurino for their support in carrying out the implementation of LoCoMoTiVe. This work was partially supported by the University of Salerno (grant number 300392FRB16COSTA).

Appendix A. XML Entity-relationship diagram specification

```
<language name="EntityRelationship">
    <element ref="ENTITY">
        <property name="$Key" type="list&lt;string>" postcondition="size($Key)>0">
            <function name="add">
                <param>CON_E_A/KEY_ATTR</param>
                <param>@KeyName</param>
            </function>
        </property>
        <property name="$Attributes" type="list&lt;string>">
            <function name="add">
                <param>CON_E_A/ATTRIBUTE</param>
                <param>@AttrName</param>
            </function>
            <function name="add">
                <param>CON_E_R[@Cardin='([01],1)']/RELATION</param>
                <param>$Attributes</param>
            </function>
        </property>
        <action><![CDATA[
            print(@EntName + " (<u>" + explode(", ", $Key) + "</u>);
            if(size($Attributes) > 0) print(", " + explode(", ", $Attributes));
            print(")\n");}
        ]]></action>
    </element>
    <element ref="RELATION">
        <property name="$Key" type="list&lt;string>">
            <function name="addAll">
                <param>CON_E_R[@Cardin='([01],N)']/ENTITY</param>
                <param>$Key</param>
            </function>
        </property>
        <property name="$Attributes" type="list&lt;string>">
            <function name="add">
                <param>CON_R_A/ATTRIBUTE</param>
                <param>@AttrName</param>
            </function>
        </property>
        <property name="$Entities" type="list&lt;string>">
            <function name="add">
                <param>CON_E_R[@Cardin='([01],N)']/ENTITY</param>
                <param>@EntName</param>
            </function>
        </property>
        <property name="$KeyAttrs" type="list&lt;string>" postcondition="size($KeyAttrs)=0">
            <function name="add">
                <param>CON_E_A/KEY_ATTR</param>
                <param>@KeyName</param>
            </function>
        </property>
        <action><![CDATA[
            if(size($Entities == 2)) {
                print($Name + " (<u>" + explode(", ", $Keys)) + "</u>");
                if(size($Attributes) > 0) print(", " + explode(", ", $Attributes));
                print(")\n");
            }
        ]]></action>
    </element>
</language>
```

References

[1] Matsuzawa Y, Tanaka Y, Sakai S. Measuring an impact of block-based language in introductory programming. In: Brinda T, Mavengere N, Haukijarvi I, Lewin C, Passey D, editors. Proceedings of the stakeholders and information technology in education and IFIP advances in information and communication technology, 493. Cham: Springer; 2016. p. 16–27.

[2] Costagliola G., De Rosa M., Fuccella V., Local context-based recognition of sketched diagrams, J Vis Lang Comput 25 (6) (2014) 955–962, doi:10.1016/j.jvlc.2014.10.021, (Distributed Multimedia Systems {DMS2014} Part I), http://www.sciencedirect.com/science/article/pii/S1045926X14001141.

[3] Costagliola G, De Rosa M, Fortino V, Fuccella V. RankFrag: a machine learning-based technique for finding corners in hand-drawn digital curves. J Vis Lang Sentient Syst 2015a;1:29–38. doi:10.18293/VLSS2015-043.

[4] Libkin L, Martens W, Vrgoč D. Querying graph databases with XPATH. In: Proceedings of the sixteenth international conference on database theory, ICDT '13. New York, NY, USA: ACM; 2013. p. 129–40. doi:10.1145/2448496.2448513.

[5] Golin EJ. Parsing visual languages with picture layout grammars. J Vis Lang Comput 1991;2(4):371–93. doi:10.1016/S1045-926X(05)80005-9.

[6] Rekers J, Schurr A. A graph based framework for the implementation of visual environments, visual languages. In: Proceedings of the IEEE symposium on visual languages; 1996. p. 148–55. doi:10.1109/VL.1996.545281.

[7] Bardohl R, Minas M, Taentzer G, Schürr A. Handbook of graph grammars and computing by graph transformation. In: Application of graph transformation to visual languages. River Edge, NJ, USA: World Scientific Publishing Co., Inc.; 1999. p. 105–80. doi:10.1006/jvlc.1996.0027.

[8] Costagliola G, Polese G. Extended positional grammars. In: Proceeding of the IEEE international symposium on visual languages; 2000. p. 103–10. doi:10.1109/VL.2000.874373

[9] Marriott K. Parsing visual languages with constraint multiset grammars. In: Hermenegildo M, Swierstra S, editors. Programming languages: implementations, logics and programs. Lecture notes in computer science, 982. Berlin Heidelberg: Springer; 1995. p. 24–5. doi:10.1007/BFb0026810.

[10] Weitzman L, Wittenburg K. Relational grammars for interactive design, visual languages. In: Proceedings of the IEEE symposium on visual languages; 1993. p. 4–11. doi:10.1109/VL.1993.269572.

[11] Zhang DQ, Zhang K. Reserved graph grammar: a specification tool for diagrammatic VPLs. In: Proceedings of the IEEE symposium on visual languages (Cat. No.97TB100180); 1997. p. 284–91. doi:10.1109/VL.1997.626596.

[12] Marriott K, Meyer B. On the classification of visual languages by grammar hierarchies. J Vis Lang Comput 1997;8(4):375–402. doi:10.1006/jvlc.1997.0053. http://www.sciencedirect.com/science/article/pii/S1045926X97900537

[13] Costagliola G, Deufemia V, Polese G. A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans Softw Eng Methodol 2004;13(4):431–87. doi:10.1145/1040291.1040293.

[14] Costagliola G, Deufemia V, Polese G. Visual language implementation through standard compiler–compiler techniques. J Vis Lang Comput 2007;18(2):165–226. doi:10.1016/j.jvlc.2006.06.002. http://www.sciencedirect.com/science/article/pii/S1045926X06000371

[15] Bardohl R. GENGED: a generic graphical editor for visual languages based on algebraic graph grammars, visual languages, 1998. In: Proceedings of the IEEE symposium on visual languages; 1998. p. 48–55. doi:10.1109/VL.1998.706133.

[16] Chok SS, Marriott K. Automatic construction of intelligent diagram editors. In: Proceedings of the eleventh annual ACM symposium on user interface software and technology, UIST '98. New York, NY, USA: ACM; 1998. p. 185–94. doi:10.1145/288392.288603.

[17] Minas M, Viehstaedt G. DiaGen: a generator for diagram editors providing direct manipulation and execution of diagrams. In: Proceedings of the eleventh international IEEE symposium on visual languages, VL '95. Washington, DC, USA: IEEE Computer Society; 1995. p. 203. http://dl.acm.org/citation.cfm?id=832276.834276

[18] Zhang DQ, Zhang K. VisPro: a visual language generation toolset. In: Proceedings of the IEEE symposium on visual languages (Cat. No.98TB100254); 1998. p. 195–202. doi:10.1109/VL.1998.706163.

[19] de Lara J, Vangheluwe H. AToM3: a tool for multi-formalism and meta-modelling. In: Kutsche RD, Weber H, editors. Fundamental approaches to software engineering. Lecture notes in computer science, 2306. Berlin Heidelberg: Springer; 2002. p. 174–88. doi:10.1007/3-540-45923-5_12.

[20] Kastens U, Schmidt C. VL-Eli: a generator for visual languages – system demonstration.. Electr Notes Theor Comput Sci 2002;65(3):139–43. http://dblp.uni-trier.de/db/journals/entcs/entcs65.html#KastensS02

[21] Schmidt C, Kastens U, Cramer B. Using devil for implementation of domain-specific visual languages. In: Proceedings of the workshop on domain-specific program development; 2006. p. 38.

[22] Zhang KB, Orgun MA, Zhang K. Visual language semantics specification in the VisPro system. Proceedings of the selected papers from the Pan-Sydney workshop on visualisation, 22. Darlinghurst, Australia: Australian Computer Society, Inc; 2002. p. 121–7. http://dl.acm.org/citation.cfm?id=1164094.1164115

[23] Costagliola G, De Rosa M, Fuccella V. Extending local context-based specifications of visual languages. J Vis Lang Comput 2015b;31(Part B):184–95. doi:10.1016/j.jvlc.2015.10.013. Special Issue on {DMS2015}, http://www.sciencedirect.com/science/article/pii/S1045926X15000701