Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the
Third International Workshop on Graph Based Tools
(GraBaTs 2006)

AGraphs: Definition, implementation and tools

David Déharbe, Anamaria Martins Moreira and Demóstenes Sena

13 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

AGraphs: Definition, implementation and tools

David Déharbe[1], **Anamaria Martins Moreira**[1] **and Demóstenes Sena**[1]

[1] Universidade Federal do Rio Grande do Norte — UFRN
Natal, RN, Brazil
david,anamaria,demost@dimap.ufrn.br

Abstract: AGraphs are a graph-based language representation, transformation and exchange format. In the same vein as XML, AGraphs form a general data representation mechanism that needs to be instantiated in different specific applications. In this paper, we present the AGraphs data structure, programming interface and related tools, identify their main features with respect to exchange format characteristics, and compare them to other existing exchange formats. These different features are illustrated on an instance of AGraphs for modular Petri nets.

Keywords: language representation, graph formats

1 Introduction

Tools that manipulate programs or specifications need some internal representation of the information being manipulated as well as means of exchanging this information with other tools or among its own components. Some representation formats have been proposed and used in different tools. Of course, different needs tend to direct researchers and developers into different kinds of formats, usually terms or graphs [2, 19, 10]. In [14] a graph based representation for the algebraic specification language CASL [1] is proposed, but this representation format is indeed much more general and can be used in the representation of different languages in different contexts. Graph nodes and vertices represent respectively program elements and relations thereof. Moreover, nodes have attributes representing information specific to the corresponding program elements. We call this generic representation data structure AGraphs; a language (be it programming, specification or hardware description) is handled by a specific AGraph instance.

The first AGraph instances were developed manually to represent hardware descriptions in a VHDL-based model checker [6] and algebraic specifications in two instances of the development tool FERUS [14, 15]. The common characteristics of these first three instances were then formalized and led to the definition of an AGraph schema language, and to the development of AGraph instance generators for C and JAVA.

An AGraph schema as well as its API are language dependent, i.e., they are tailored to the information being represented. One of the qualities of this approach is that it results in quite simple and intuitive APIs, making the AGraph representation very easy to program with. AGraphs also provide textual and binary file formats for compressed or uncompressed archival and data exchange. The AGraph data files can be accessed by both C and JAVA without compatibility issues.

Plan of the paper. The rest of the paper is organized as follows. Section 2 presents an example of an *ad hoc* language to describe modular Petri nets that will be used to illustrate AGraph concepts throughout the paper. Section 3 presents AGraphs, their specification, design, interface, implementation. Section 4 describes the available tool support for developers wishing to employ AGraphs. Related work is discussed in Section 5, and conclusions and future work are presented in Section 6.

2 Example

We present an ad-hoc modular Petri net description language [12] that will be used as a running example throughout the paper. For instance, consider the Petri net of Figure 1. The main net is N1; it has one place import (i1), three places (p1, p2, p3) and three transitions (t1, t2, t3). Places p2 and p3 are exported. N1 instantiates a net N2, by instantiating N2's import place with place p1. The place o exported by N2 is an input place of transition t3. There is one token in place p1 and none elsewhere. The corresponding textual description in our description language would be as in Figure 2. The syntax of the language can be described in the SDF formalism [17], but this description is omitted by lack of space.

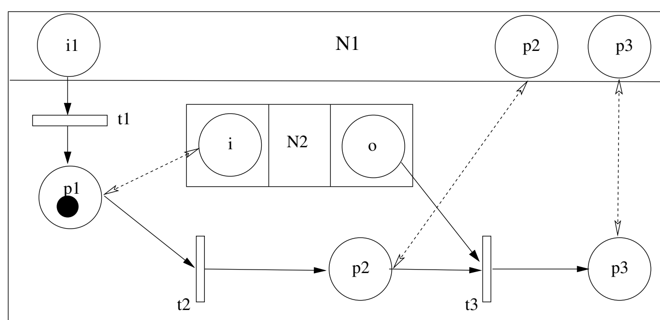

Figure 1: A simple Petri net N1 with an embedded net N2

3 AGraphs

The development of graph-based libraries to represent and manipulate structured texts (programs and specifications) arose in the development of two different projects (a VHDL parser [6] and an algebraic specification integrated development environment [14]) where they proved to be adequate to implement modeling and transformation operations for program development tools.

Those libraries evolve around the same data structuring and interfacing concepts. These common concepts form the AGraph representation and manipulation framework and can be reused for most, if not all, programming and specification languages. AGraphs are therefore a suitable framework to apply generic graph transformation techniques to, e.g., implement software evolution rules. The concepts underlying AGraphs are presented in the following sections.

```
net N1 =
  places p1, p2, p3
  subnets N2 (i:p1)
  imports i1
  exports p2, p3
  transitions
    t1 : i1 -> p1
    t2 : p1 -> p2
    t3 : N2.o, p2 -> p3
  tokens p1:1, p2:0, p3:0
```

Figure 2: Textual description of the Petri net N1 pictured in Figure 1

3.1 Design of the AGraph representation

All modern program or specification languages have a notion of compilation unit which can reference other units and can be compiled (or analyzed) independently from other units (although it may rely on some other units being already compiled). Moreover, in some languages, collections of related modules can be grouped into libraries. In general, a unit in a library may reference entities defined in another unit stored in a different library.

In the AGraph format, each unit is represented by a set of interconnected *nodes*. Roughly, each node is associated with a unit and represents an element of the abstract syntax of this unit. There are therefore several *kinds* of nodes, each corresponding to a non-terminal of the abstract grammar of the represented language. For instance, for our Petri net example, places, transitions and the net module itself, would be represented by three different kinds of nodes, say `place`, `transition` and `net`.

Each kind of node has a different signature, composed of typed and labeled *attributes* and (outgoing) *edges* and *hyperedges* targeting other nodes. Attributes are atomic unstructured values. For instance, a net node has an attribute labeled `aIdentifier`, storing its identifier, and several outgoing hyperedges pointing towards the different components of the net, such as places and transitions (hyperedges connect a node to a list of nodes).

The rationale of the AGraph format is then that the representation of units (e.g. Petri nets) and libraries thereof are represented as a directed graph, where vertices correspond to entities in the units, and edges to relationships between those entities.

3.2 Interface provided for an AGraph representation

An AGraph implementation is a library accessible through the AGraph application programming interface (API). The interface provided by AGraphs exposes the directed graph metaphor to client applications but hides all the implementation details. The functions provided by the API are: (1) a library initialization routine; (2) a constructor for each kind of node; (3) two accessors that respectively provide query (get) and assignment (set) functionalities for each possible type of attribute and edge; (4) input (resp. output) functions to read (resp. write) a unit from (resp. to) file; (5) a compression (resp. decompression) function to create a file in the binary (resp. textual)

format from a file in the textual (resp. binary) format.

3.3 Schema specification

As explained in Section 3.1, AGraphs contain different kinds of nodes, node attributes and edges. The general structure of AGraphs, presented in Section 3.1, has then to be instantiated for each language to be represented. Such instantiations are actually implemented to represent language units. An instantiation is defined by an AGraph *schema*, defining the available nodes, their structure (signature), their relations and, indirectly, the corresponding API functions. In this section, we describe the specification of AGraph schemas.

The node representing a unit in a given language is called *RootNode* in an AGraph schema. The specification of the AGraph schema defines which among the language specific nodes is root. It also specifies a label that identifies the language being represented and is prepended to the AGraph identifiers. This label should be unique in an environment, so that if different AGraph instantiations co-exist in the environment there won't be name clashes.

The kinds of nodes of the instantiation described by the schema are then presented. Some of them have specific descriptions, as is the case with *root nodes*, *list nodes* and *import nodes*, but all node kinds are identified by a name and have specific basic *node components*. There are three types of node components:

- *Attributes* maintain information of primitive (e.g., integer) or user defined enumerated types. Each node attribute has its name and type defined in the schema.
- *Edges* link the node to another node. The schema description of an edge contains its name and the set of identifiers of the kinds of nodes to which an instance of this edge can point to.
- *Hyperedges* link the node to a sequence of nodes. The schema description of a hyperedge contains its name and the name of a list kind of nodes through which the different values will be linked to the edge.

The list nodes represent hyperedges. The schema description of a list kind of nodes identifies three edges, pointing respectively to the information node, the next node, and the previous node.

The schema description of the root kind of node indicates the attribute that contains the identifiers of the represented units and the hyperedge pointing to the root nodes of the imported units.

The import node kind contains information about imported units. These nodes contain common and specific components. The schema description defines the name of the node kind and an edge to the root node of the imported unit.

3.4 Implementation of AGraph libraries

The implementation of the above concepts is based on register data structures and pointers in a classical imperative language such as C and on objects and object references in an object oriented language such as JAVA. In this section we provide some information on how an AGraph instance is implemented.

3.4.1 Implementation of the graph nodes

An example of the definition of the structure of a node is given in Table 1. This node corresponds to a Petri net token specification (token), with the corresponding place (nPlace) and quantity (aQuantity). The first four fields are mandatory AGraph fields (i.e. they are present for all kinds of nodes and for any language representation), whereas the remaining correspond to data specific to the represented semantic domain (Petri nets in this example).

C implementation:	**JAVA implementation:**
typedef struct { pnRef position;	public class token {// class variables
void * aToolInfo;	pnRef position;
pnKind aKind;	Object aToolInfo;
pnRef nRoot;	pnKind aKind;
pnRef nPlace;	pnRef nRoot;
int aQuantity;	pnRef nPlace;
} token;	int aQuantity;
	// class methods...}

Table 1: Examples of the definition of a node kind in C and Java

3.4.2 Unit dependencies

Our Petri net language is modular and supports hierarchical net composition. To reference external modules is common place in most languages. An efficient representation shall not fully instantiate external modules in the representation, but provide instead mechanisms to refer to and access the representation of these modules. To support such interdependency, AGraphs include a mechanism to refer to external units, i.e. an edge (or a hyperedge) may point to a node outside the current unit. As a consequence, when a unit is loaded into memory, the imported units are recursively loaded. Note that the algorithm implementing this dynamic loading feature avoids duplicate representation of units.

The internal state of an AGraph library contains the graph representing the nodes of the units loaded into memory as well as additional data needed to handle cross references between units. Each unit has a unique identifier (a natural number) and its representation is composed of two tables: the first contains the identifiers of the imported units, while the second stores the subgraph representing the unit. The file formats of AGraphs are also based on those two tables.

3.4.3 AGraph API

The API of an AGraph library has the elements described in Section 3.2.

The constructors for each kind of node create the node in memory using the values passed in its parameters. Note that some of the parameters may be undefined at creation time. In this case, they can be given a default value, and can be later assigned using accessors provided by the library. For instance, the declarations of the constructors in C and JAVA for the node of Table 1 are:

`pnNode pnMakeToken (pnNode Place, int Quantity);`, in C, and
`public Token (pnNode Place, int Quantity);`, in JAVA

Also, for each different attribute or edge label of each node, there are corresponding query and assignment functions. In C, each query function has a unique parameter (the queried node) and, as return type, that of the corresponding attribute or edge. In JAVA, the queried node is the object that invokes the query function and does not need parameter. Each assignment routine, in C, has two parameters: the node being modified, and the new value for the modified field. In JAVA, the assignment routine has a single parameter: the new value for the modified field. Examples of accessor declarations are:
`pnNode pnGetPlace(pnNode n);`, in C, and
`public pnNode pnGetPlace();`, in JAVA.

3.4.4 Persistent formats for AGraphs

Each AGraph instance has corresponding binary and text-based formats used for file-based storage and exchange. The binary format stores a custom compression of the textual format. The textual format is composed of a header and structured elements. Each structured element stores one node, and is a sequence of integers and strings, as in:

```
<0 1 2 N1 0 0 0 0 0 0 >
```

which represents an empty Petri Net with name `N1` in an AGraph derived from the schema presented in the next section. The 6 positions to the right of `N1` serve to indicate the position in the sequence of elements corresponding to the 6 hyperedges defined in the schema to represent the different items of a Petri Net.

Additionally to this AGraph specific format, AGraph libraries contain a function to generate a GXL [19] output representantion of the graph. Through the GXL representation, AGraphs may benefit from GXL tools, as for instance, graph visualization.

4 Automated generation of AGraph libraries

The implementation of AGraph libraries follows a certain number of conventions. This set of strict conventions makes it easy, but cumbersome, to code a new AGraph library. We have thus implemented two methods to automatically generate AGraph library implementations:

- In the first method, the user describes the different kinds of nodes in an XML-based schema description language. This approach is presented in Section 4.1. The main advantage of this method is its flexibility, at the expense of requiring additional work from the user.

- In the second method, the representation schema is automatically generated from a description of the syntax of the language. The advantage of this approach is that it can be generated from the same description that would be used to generate a parser (with possibly some extra annotations) and the burden of designing a schema for the language is avoided. Details are provided in Section 4.2.

4.1 Customized generation

The first input format proposed for our generator is an XML-based description of its nodes and some additional information concerning, e.g., the identification of the root of the graph. Because the set of operations to manipulate the graphs is fixed for each kind of node (a `make` function for the node and a `set` and a `get` for each attribute of the node) this information is enough to automatically generate the AGraph library.

To describe the schemas for a given programming or specification language, we defined a simple XML elements syntax, described below.

Customized input format specification The XML-based schema description language has 9 different elements: Unit, Datatype, Attribute, Edge, Hyperedge, Node, RootNode, Import and List.

`Unit` is the main element delimiting the document. It contains XML-attributes[1] identifying the language being represented (XML-attribute `lang`) and the node kind of the graph root (XML-attribute `root`), e.g.:

```
<Unit lang="pn" root="net" >
```

The `Unit` element contains a sequence of `DataType` elements, followed by exactly one `RootNode` element, and any number of `Node`, `Import` and `List` elements.

The `DataType` element may be used to declare named enumerated data types to be later employed to define node attributes. For instance, assume that we want to represent a net with colored tokens, where the color may be white, grey or black, we can declare the corresponding data type:

```
<DataType name="TokenColor" value="White|Grey|Black" />
```

The `Node` element specifies a node kind, identified by its name (XML-attribute `name`), and embeds elements that can be either `Attribute`, `Edge` or `Hyperedge`.

The `Attribute` element specifies the characteristics of a node attribute. It has two mandatory XML-attributes: `name`, to identify the attribute, and `type` to specify the type of the attribute values. For example, nodes that represent nets will have an attribute to represent their name:

```
<Attribute name="aIdentifier" type="string" />
```

`Edge` specifies the characteristics of an edge. It has two mandatory XML-attributes: `name`, to identify the edge, and `type` to specify the legal types of the edge destination. For example, nodes representing tokens will have an edge towards the node of the corresponding place:

```
<Edge name="nPlace" type="place" />
```

`Hyperedge` specifies the characteristics of a hyperedge (edge with several destination nodes). It inherits the XML-attributes of `Edge`, with the further restriction that the `type` XML-attribute shall be the identifier of a list node kind. For example, nodes representing nets will have a hyperedge towards the list of places in the net:

[1] To avoid confusion between AGraph node attributes and XML element attributes, we call the latter *XML-attributes*.

```
<Hyperedge name="nPlaces" type="placelist" />
```

The elements `Attribute`, `Edge` and `Hyperedge` are then combined to define a node kind, as in:

```
<Node name="token">
        <Edge name="nPlace" type="place" />
        <Attribute name="aQuantity" type="int" />
</Node>
```

The `RootNode` element specifies the node kind of the root. It is a specialization of the `Node` element, and must additionally include a `rootIdentifier` XML-attribute, that specifies the name of the attribute that contains the unit name, and a `listImports` XML-attribute, that specifies the name of the edge accessing the list of imported units (when relevant). For example, in our modular Petri net language, this element would be:

```
<RootNode name="net"
      rootIdentifier="aIdentifier"
      listImports="nSubnets">
  <Attribute name="aIdentifier" type="string">
  <Hyperedge name="nPlaces" type="placelist">
  <Hyperedge name="nSubnets" type="subnetlist">
  <Hyperedge name="nImports" type="placelist">
  <Hyperedge name="nExports" type="bindinglist">
  <Hyperedge name="nTransitions" type="transitionlist">
  <Hyperedge name="nTokens" type="tokenlist">
</RootNode>
```

The `Import` element specifies an external unit reference mechanism. It is a specialization of the `Node` element with an additional XML-attribute `rootIdentifier` specifying the edge pointing to the root of the imported unit. The XML-attribute value must match that of an existing sub-element. In our example, we have a single external reference mechanism, which is the subnet:

```
<Import name="subnet" rootIdentifier="nSubnet">
        <Edge name="nSubnet" type="net" />
        <Hyperedge name="nPorts" type="bindinglist" />
</Import>
```

Finally, the `List` element specifies a list of nodes. It has four mandatory XML-attributes to identify the name of the node class, the name of the edges leading to the value, the successor and the predecessor of the current cell. Additional edges and attributes or constraints on mandatory edges may also be specified:

```
<List name="bindinglist" next="nNext" previous="nPrevious" value="nValue">
        <Edge name="nValue" type="binding" />
</List>
```

4.2 Generation from syntax descriptions

We also provide a direct connection from the syntax definition of a language L (e.g. a Petri net description language) to the corresponding graph format. This connection shows that it is

possible to extract the abstract graph representation from a description of the concrete syntax. Having this mapping automated allows us to: (1) use the syntax definition as documentation of the graph format; (2) quickly prototype new formalisms; and (3) react efficiently to new versions of a formalism.

We reuse an existing tool that maps syntax definitions to abstract data-types [11]. From the abstract data-type generated by this tool, an AGraph format definition can be effectively generated. The generation process follows a linear flow: (1) the primary input is a syntax definition in SDF [17], from which two artifacts may be generated: a parser and a so-called Abstract Data Type Definition file; (2) the Abstract Data Type Definition file is translated to an AGraph format definition (using the schema presented in Section 4.1), by rephrasing the original tree-oriented definition to a graph-oriented definition; (3) this graph definition is then used to generate an AGraph library in C and/or JAVA.

The artifacts produced in this generation process may be employed to write a compiler for *L* (or any kind of transformation tool). The parser resulting from the first step may be used to generate the parse tree of any program in the language *L* (e.g. a Petri net description), and the AGraph library may be used to construct an attributed graph from this parse tree. The overall architecture of this flow is represented in Figure 3.

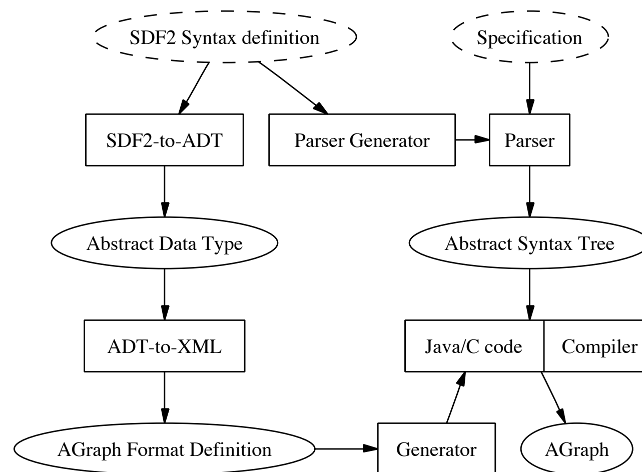

Figure 3: AGraph generation and deployment

This architecture introduces a close correspondence between the syntax of a module in the defined formalism, and the corresponding AGraph representation of the module. The Abstract Data Type Definition documents the contract between the parser and the compiler, and the AGraph format definition documents the contract between the compiler and other tools that will manipulate a module of *L*.

5 Related work

AGraphs are primarily designed for the representation and manipulation of programming and specification languages. As they are based on a graph data structure, they can also be viewed as a graph-representation language. AGraphs are a generic data structure that can be instantiated for specific domains. A significant effort has been made to automate the implementation of each instance from high-level specifications (grammars or graph schemas).

In the following, we discuss the relationship of AGraph with other important graph formats and compare the automated generation capabilities against that of other (non graph-based) formats.

5.1 Graph formats

Several graph description languages and graph-based exchange formats have been proposed so far for specific applications [16, 9] or with the intention of becoming defaults in some application domain [7, 3, 19]. Some of them, such as GDL [16] concentrate on visual characteristics of the graph being described, while others, such as GraphML [3] or GXL [19] are more flexible and general, and may be considered as standards in their respective domains.

GraphML, oriented to graph drawing, is an extensible and flexible graph-description language that provides mechanisms to describe the structure of graphs, as well as to include application-specific annotations. GXL, a standard for software reengineering tools, is also flexible and generic, subsuming many other existing graph formats. In the same vein as in the work presented in [4], an AGraph could be translated into GraphML for visualization. Also, GXL could be used as a more verbose AGraph persistent format, extending GXL-based tools to AGraphs.

On the other hand, AGraphs provide facilities for generating graph instances from programs or specifications and to manipulate these graph instances. Thus, AGraphs characterise themselves as more than just an exchange format, also providing corresponding support tools. This, as far as we know, is out of the scope of most available formats.

A possible exception to this is the Meta Object Facility (MOF [2]) with, e.g., its MetaData Repository (MDR [3]) implementation. However, although extremely powerful tools, and because of that, they are also very verbose, require a bigger initial investment for developing an application and tend to generate much bigger source code.

5.2 Automated generation

Code generation is an effective, and increasingly popular, method for software engineering [5]. The AGraph generator was developed in the tradition of code generators that take data-type definitions as input. There are many such systems that generate implementations for tree-like data-structures, e.g. ApiGen [11], ASDL [18], JJForester [13], and JAVA Tree Builder. Also, code generators such as Breeze XML Studio for binding XML files to an object hierarchy are based on similar principles.

The main advantage of AGraphs with respect to these works is that the generated code implements a graph data-structure instead of a tree. Since cross-references are very common in

[2] `http://www.omg.org/mof`

[3] `http://mdr.netbeans.org`

applications that analyze and transform programs or specifications, AGraphs immediately offer the data-structure that is actually needed during computation. This results in efficient and compact code. Tree-based data-structures need to be complemented with for example hash-tables to accomplish the same cross-referencing ability.

6 Conclusions and Future work

AGraphs are a graph-based language representation and data exchange format. They form a general data representation mechanism to be instantiated for each language to be represented. This characteristic, fully explored in its API, makes AGraphs simple to use. For instance, to get the value of a node attribute called `X`, a function `langGetX` is to be called (where `lang` is a prefix given by the user at generation time). This approach differs from other known graph manipulation libraries, such as libGR, used in the Gr-Gen tool [8], which are usually generic.

Automated generation of the implementation of an AGraph instance in the languages C and JAVA can be performed from a schema defined in a XML format or from a grammar in SDF. The file format can be used as a communication mechanism between tools built with both C and JAVA instances of AGraphs. The AGraph generator can be extended to support other programming languages.

Both manually and automatically generated AGraph instances have been effectively used in-house in different software development efforts and have proved an easy and practical mean to represent, manipulate and transform software artifacts (specifications and programs).

As future work, we plan to improve interoperability by taking several actions: port the generator to other programming languages whenever needed, establish a bridge with GraphML and GXL. Currently, the AGraph generated library contains a function to write a GXL representation of the graph. It is also possible to generate a function that reads into memory a GXL representaion of an AGraph. This approach would loose in file size but would gain in interoperability. We also propose to improve programming support by adding visualization tools. Finally, we also plan to use AGraphs to support other research projects related to programming languages and software engineering.

Bibliography

[1] M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 (IFIP Series) of *Lecture Notes in Computer Science*. Springer Verlag, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

[2] M. Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[3] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. Graphml progress report: Structural layer proposal. In *Proceedings 9th International Symposium on Graph Drawing (GD '01), Springer Lecture Notes in Computer Science 2265*, 2002.

[4] U. Brandes, J. Lerner, and C. Pich. GXL to GraphML and vice versa with XSLT. *ENTCS*, 127:113–125, 2005.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] D. Déharbe, S. Shankar, and E. Clarke. Model checking VHDL with CV. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 508–514. Springer Verlag, 1998.

[7] J. Ebert, B. Kullbach, and A. Winter. Grax – an interchange format for reengineering tools. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 89–98, 1999.

[8] R. Geiss, G. Batz, D. Grund, S. Hack, and A. Szalkowski. Gr-Gen: A Fast SPO-Based Graph Rewriting Tool. In *Proceedings of the Third International Conference on Graph Transformations*, volume 4178 of *LNCS*, pages 383–397, 2006.

[9] M. Himsolt. Gml, a graph modelling language, 1997. `http://infosun.fmi.uni-passau.de/GraphLet/GML/`.

[10] D. Jin. Exchange of software representations among reverse engineering tools. Technical report, Department of Computing and Information Science - Queens University, December 2001.

[11] H. d. Jong and P. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59, 2004.

[12] E. Kindler and M. Weber. A universal module concept for petri nets: an implementation-oriented approach. Informatik Bericht 150, Institut für Informatik, Universität zu Berlin, 2001.

[13] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[14] A. M. Moreira, C. Ringeissen, D. Déharbe, and G. Lima. Manipulating algebraic specifications with term-based and graph-based representations. *Journal of Logic and Algebraic Programming*, 59:63–87, 2004.

[15] A. M. Moreira, C. Ringeissen, and A. Santana. A Tool Support for Reusing ELAN Rule-Based Components. *Electronic Notes in Theoretical Computer Science*, 86(2), 2003.

[16] G. Sander. Vcg – visualization of compiler graphs. Technical report, Universität des Saarlandes, February 1995.

[17] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sep. 1997.

[18] D. Wang, A. Appel, J. Korn, and C. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[19] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL Graph Exchange Language. In S. Diehl, editor, *Revised Lectures on Software Visualization International Seminar*, number 2269 in LNCS, pages 324–336, London, UK, 2002. Springer-Verlag.