Electronic Notes in Theoretical Computer Science 1 (1995) *to appear*

DIEGO, Another Step Towards a Module Concept for Graph Transformation Systems

Gabriele Taentzer [a,1], Andy Schürr [b]

[a] *Computer Science Department, TU Berlin, D-10587 Berlin, Germany*

[b] *Lehrstuhl für Informatik III, RWTH Aachen, D-52074 Aachen, Germany*

Abstract

DIEGO module systems offer means for DIstributed programming with Encapsulated Graph Objects. Their operational semantics definition follows the lines of distributed graph transformation, thereby permitting the specification of distributed systems with concurrently interacting objects. Adapting previously made proposals for a graph grammar module concept and for graph models with information hiding to this setting, the new approach supports structuring of large specifications into small reusable modules with well-defined interfaces between them. Use relations between export interfaces and import interfaces are our means to construct system architectures and to allow reuse of modules in different environments.

1 Introduction

After 25 years of graph grammar research we are nowadays able to specify small software packages by means of *monolithic graph transformation systems*. Using a graph grammar programming language like PROGRES [5], we are even able to interpret these specifications directly and to translate them into efficiently executable ordinary programs. But as soon as we are trying to deal with "real world" problems, we have to realize that

- there is almost no support for breaking big monolithic specifications into reusable small pieces with well-defined interfaces between them and
- there is still insufficient support for modeling distributed systems of complex structured and concurrently (inter-)acting graph objects.

A very *abstract framework* is given in [1], where several concepts for modules needed are mentioned, but without introducing a concrete graph data model and a specific graph rewriting approach. The concept of *transformation units* of [3] has its main focus on grouping sets of related rewrite rules and

[1] This work has been partly supported by the ESPRIT Working Group 7138 "Computing by Graph Transformation (COMPUGRAPH II)"

modeling *use* relations between these sets; it neglects aspects like distribution, inheritance, and information hiding of graph structures.

Furthermore, there are those two approaches which form the base for our new formalism, so-called *DIEGO module systems*: hierarchically **DI**stributed graph transformations [6] and hierarchical **E**ncapsulated **G**raph **O**bjects [2]. The first approach has its main focus on distribution, whereas the second approach studies extensively information hiding, inheritance, and import/export relations for hierarchical graphs and graph types. Both approaches are more or less complementary to each other, but we believe that they complete each other perfectly.

The new formalism presented here may be regarded as a first instantiation of some of the ideas presented in [1]. It proposes a concrete graph data model and an operational semantics for graph rewrite rules (in the DPO-approach). It contains the ideas of *modules with import/export interfaces*, but here with a tight (concrete) operational semantics. Furthermore, our new formalism offers support for *distributed programming* by defining transitions from one distributed system state to another one as a particular DIEGO module system's operational semantics.

Example 1.1 (Distributed SEE Data Repository) For illustrating our new approach we will use a small example which is a slightly modified and simplified version of the running example in [6]. It is the specification of a distributed software engineering environment's (SEE's) data repository. Such a repository is a set of interrelated workspaces, where each workspace resides on a specific machine (within a local area network). In the simplest case, any project has a *single public workspace*, which contains all (released versions of) documents. Any software developer assigned to a particular project has an *own private workspace*, which contains copies of all needed released document versions as well as new private successor versions. An operation *Checkout* copies latest versions of needed documents from the public workspace to a given private workspace and locks these versions in the public workspace for further usage. *Checkin* copies modified document versions from the private workspace back to the public workspace without releasing the lock, which is the task of another operation *ReleaseLock*. Further operations are needed for creating new documents, etc. but omitted over here due to lack of space.

The specification of such a scenario requires the introduction of two subspecifications, one describing the behavior of public workspaces, the other one describing the behavior of private workspaces. Both types of workspaces contain data (nodes and edges) which should not be visible for the outside world. Private workspaces, for instance, may not export knowledge about currently stored versions of documents. Therefore, *information hiding* of data as well as local operations on data must be supported. Furthermore, we must be able to model the fact that the subspecification for private workspaces *uses* the subspecification for public workspaces for realizing its own operations.

2 Systems of Encapsulated Graph Object Modules

When developing a system of modules we want to distinguish between the instance level and the specification level. On the *instance level*, we have a distributed system of graphs, where each graph hides parts of its internal structure and shares knowledge about exposed parts with other graphs. Therefore, we call these instances **E**ncapsulated **G**raph **O**bjects (EGOs). On the *specification level*, we have a system of related subspecifications, where each subspecification is a kind of graph transformation system with an import/export interface to other subspecifications. Each subspecification defines the behavior of a certain class (type) of EGOs and is therefore called *EGO module*.

The following concepts are based on is the notion of a *graph transformation system GTS* which consists of a start graph $S(GTS)$ and a set of productions $P(GTS)$ in the simplest case. Since different graph transformation systems are set into relation later on, we need the concept of a GTS morphism. Here, the simplest representative is a pair $f = (s(f), p(f))$ of mappings, one for the *start graphs* and one for the *sets of local productions*. $s(f)$ is just a graph morphism and $p(f)$ is a mapping between sets of productions such that its arguments are subproductions of their results (cf. [6] for the definition of subproductions).

Definition 2.1 A tuple $A = (A.imp, Aexp, A.body, A.i, A.e)$ is an **encapsulated graph object (EGO) module** with graph transformation systems $A.imp$, $A.exp$ as well as $A.body$ and GTS-morphisms $A.i : A.imp \to A.body$ and $A.e : A.exp \to A.body$.

The definition above states that an EGO module A consists of three parts: its import part $A.imp$, its export part $A.exp$, and its own realization $A.body$ which establishes the needed "implements" relation between the import and the export interface. Different EGO modules can be connected to each other at their formal import/export interfaces in the following way:

Definition 2.2 A **use relation** $u = (u.int, u.in, u.out)$ from a client EGO module A to a server module B is a graph transformation system $u.int$, called **interface**, together with GTS-morphisms $u.in : u.int \to B.exp$ and $u.out : u.int \to A.imp$, such that $p(u.out)$ maps identical productions onto each other only.

Note that the body of a client module is not directly coupled to the export part of its server module, but uses an intermediate *interface part*. In this way, specifications may be developed independently from each other, and mapping of required import resources to actually existing export resources may be done afterwards by means of use relations. Such a *use relation* determines all those productions from the server EGO module's export part which are actually needed in the client module's import part and that part of the server module's (initial) export state which is visible for all client module instances. The following example shows that client modules may *import subproductions* of the server module's exported productions and that they may extend these imported subproductions within their own bodies (but not within their import interfaces).

Definition 2.3 A **DIEGO module system** M is a graph (M^N, M^E, s_M, t_M) where the nodes of M^N are EGO modules and the edges of M^E are use relations with $s_M(u)$ indicating the server and $t_M(u)$ the client module of use relation u. Furthermore, the following consistency conditions have to be satisfied:

- There is at most one use relation between two EGO modules in each direction.
- For all nodes $n \in M^N$ and all edges $e \in M^E$ with $t_M(e) = n$ $P(n.imp)$ $(S(n.imp))$ has to be the disjoint union of all $P(e.int)$ $(S(e.int))$.

Example 2.4 (Public and Private Workspace Modules) The *public workspace* module $PuW = (PuW.imp, PuW.exp, PuW.body, PuW.i, PuW.e)$ consists of three GTS, which have an empty initial graph and all local productions of figure 1 with matching name prefixes (plus additional productions). The *private workspace* module $PrW = (PrW.imp, PrW.exp, PrW.body, PrW.i, PrW.e)$ consists of three GTS where each graph has a single $p(erson)$ node as its initial state. Their local productions are defined within figure 1, too. The morphisms of both modules are defined such that productions with same name suffixes are mapped onto each other.

The public workspace module has an empty set of imported resources and the private workspace module has an empty set of exported resources; creation and deletion of private workspaces are not modeled as local productions and, therefore, not an explicitly defined part of the export interface. The productions of both modules cooperate in the following way (cf. figure 1):

- Productions $PuW.[body \mid exp].Checkout$ give a person write access to the last version of a document by creating a new $w(rite)$ edge and making its $l(ast)$ edge invisible in the export. Both productions are executed synchronously with productions $PrW.[imp \mid body].Checkout$. This has the effect of creating a version node in the private workspace, which is bound to the still existing version in the public workspace.
- $PrW.body.Modify$ is a local production of private workspace. It creates a new private document version without affecting the state of any other EGO instance.
- Productions $PrW.[imp \mid body].Checkin$ remove the old version node from the private workspace and copy the new private version back into the public workspace. Their execution is bound to the execution of productions $PuW.[body \mid exp].Checkin$.
- $PuW.[body \mid exp].ReleaseLock$ and $PuW.[imp \mid body].ReleaseLock$ finally release the document lock and remove the still existing version node in the private workspace.

Example 2.5 (The DIEGO Module System for Data Repository) Private workspaces rely on the existence of public workspaces, and their imported

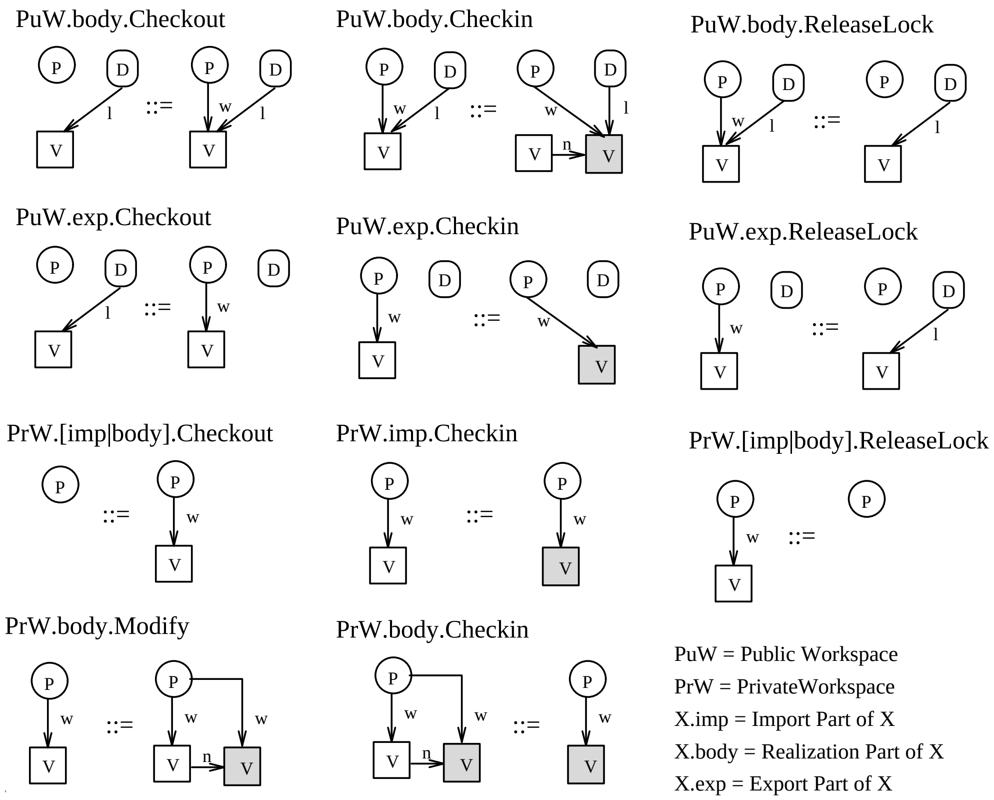

Fig. 1. Local Productions needed within the running example

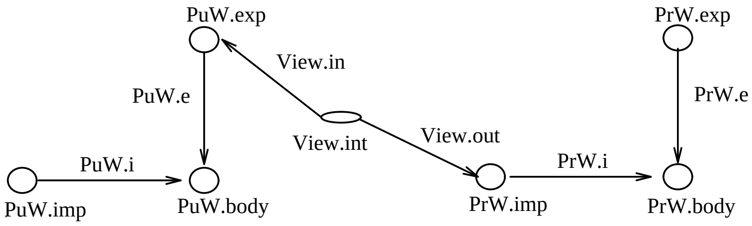

Fig. 2. A simple DIEGO Module System

productions are bound to corresponding productions of public workspaces. This is the effect of the use relation $View$ of figure 2. It has interface $View.int$ and morphisms $View.in$ as well as $View.out$ which map all productions of $PuW.exp$ onto productions with identical name suffixes in $PrW.imp$. $View.int$ itself has an identical production for any production of $PrW.imp$ and an initial graph, which contains just the regarded private workspace owner as a person node.

3 Operational Semantics of DIEGO Module Systems

The description of the operational semantics of EGO modules is based on hierarchically distributed graph transformation [2]. It distinguishes two description

[2] Note that the adjective "hierarchical" refers to the distribution relation between graphs. Hierarchical distribution means that local states resulting from a distribution process of a global state can be further distributed.

levels. On the more *abstract network level*, the topological structure of a system is described as a global graph. Such a graph contains an arbitrary number of EGO module instances, termed *encapsulated graph objects (EGO's)*, as well as use-relation instances between them, termed *knows relations*. On the more *concrete description level* the internal states of distributed EGO and use relation instances are modeled. Each EGO has a triple of local graphs as its state together with two graph morphisms between them. Each knows relation has a single local graph as its interface state together with two morphisms to the import state of its source and the export state of its target. In the beginning import, export and body of an EGO are in the initial states of the GTS specifications of their modules. Combining both description levels, the whole distributed system's state is modeled as a *hierarchically distributed graph (HD-graph)*. Each network node can be refined to its local state and each network edge shows the relations between different local states.

Definition 3.1 An **EGO** $o = (imp(o), exp(o), body(o), i(o), e(o))$ is a graph consisting of nodes $imp(o)$ for import, $exp(o)$ for export and $body(o)$ for the realization part with edges $i(o), e(o)$ from import and export to realization part. The nodes of an EGO o_A, which is an instance of an EGO module A, are labeled by $A.imp$, $A.exp$ and $A.body$, respectively. The edges $i(o)$ and $e(o)$ are labeled by $A.i$ and $A.e$, resp.

Each of the nodes n is equipped with a graph $State(n)$ and each of the edges e with a graph morphism $f_e : State(s(e)) \to State(t(e))$ where $s(e)$ is the source and $t(e)$ the target node of e.

Definition 3.2 A **knows relation** $k = (int(k), out(k), in(k))$ between EGO's o and o' consists of an interface node $int(k)$ and two edges $in(k)$ and $out(k)$ from $int(k)$ to $exp(o')$ and $imp(o)$, respectively. A knows relation k, which is an instance of a use relation u from module A to module B, connects an A EGO instance with a B EGO and its graph $int(k)$ is labeled by $u.int$. The edges $in(k)$ and $out(k)$ are labeled by $u.in$ and $u.out$, resp.

Node $int(k)$ is equipped with a state $State(int(k))$ and edges $in(k)$ as well as $out(k)$ with graph morphisms $f_{in(k)}$ and $f_{out(k)}$.

Example 3.3 (A Data Repository Instance) The HD-graph of figure 3 represents an SEE data repository with two private workspaces and one public workspace. The grey private workspace contains a checked out version and a new private version of the grey document, whereas the white workspace contains just the last public version of the white document.

Changing of states (so-called actions) can be modeled by local or distributed graph transformations, which affect either the local state of a single network node or a single EGO or even more than one EGO together with the knows relations between them. All EGO's can be handled separately due to the distribution concepts. Local or distributed actions in different EGO's and the creation of a new EGO as a global action are allowed to run true concurrently.

A certain initial HD-graph together with a set of distributed productions modeling different kinds of distributed actions form a so-called HD-graph grammar used to describe a distributed system.

Definition 3.4 The **operational semantics** of an DIEGO module system M is described by the following HD-graph grammar $Sem(M) = (S_M, C_M \cup P_M)$ with

- S_M is the empty HD-graph.
- C_M is a set of distributed productions for the creation of EGO's of the following form: The right hand side consists of a new EGO o of a module $A$with $State(imp(o)) = S(A.imp)$, $State(exp(o)) = S(A.exp)$, $State(bod(o)) = S(A.body)$, $f_{i(o)} = s(A.i)$ and $f_{e(o)} = s(A.e)$ together with all the knows relations $k \in K$ being instances of use relations to A and needed to create $State(imp(o))$, i.e. all relations $k \in K$ where $f_{out(k)}(State(int(k))$ is a subgraph of $State(imp(o))$. The left hand side just contains a node equipped with $f_{in(k)}(State(int(k)))$ for each relation $k \in K$ which can also be found in the right hand side together with the remaining parts of all $k \in K$ defined below.

  Furthermore, there is a distributed production for each use relation u in C_M. If $t_M(u) = A$ and $s_M(u) = B$ a knows relation k between an EGO o of module A and an EGO o' of module B can be established where $State(int(k)) = S(u.int)$, $f_{in(k)} = s(u.in)$ and $f_{out(k)} = s(u.out)$.
- P_M is a set of production modeling distributed actions. All those productions in a DIEGO module system which are connected with each other by production mappings form one distributed production of P_M.

The meaning of productions related to each other is that they have to be applied in a synchronized way, f.ex. when applying an exported production the

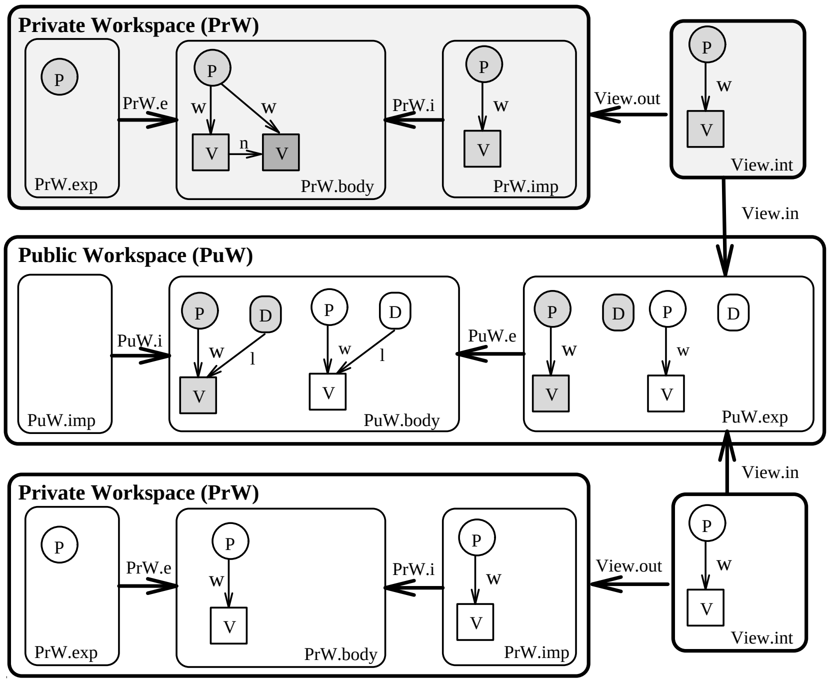

Fig. 3. Some state of a distributed data repository

corresponding body production is executed, too. Doing so, new productions of the body may become applicable. Meanwhile an import production can be applied with its corresponding body production, an interface production, an export production of another EGO module and its body production, and so on. This communication concept is in some way comparable with the well-known rendez-vous concept of synchronization.

Example 3.5 (A Data Repository State Transition) In the state of the repository described in figure 3 a checkin of the new version in the grey private workspace would be modeled by applying the productions with suffix $Checkin$ to $PuW.body$, $PuW.exp$, the grey $PrW.imp$ and $PrW.body$. Moreover, production $PrW.imp.Checkin$ is applied to the grey $View.int$. The synchronous application of productions just mentioned would lead to a state where the dark grey version is the newest of the grey document.

According to the definition of a DIEGO module system each import production has to have a counterpart in some export part. Since all productions related have to be applied synchronously, we can formulate that only those items can be imported somewhere which are exported by some connected EGO using a suitable use relation.

Proposition 3.6 *In each EGO o of module A only those parts can be imported by a production $p_{imp} \in P(A.imp)$ that are exported in a connected EGO o' of module B by production $p_{exp} \in P(B.exp)$ and p_{imp} is mapped to p_{exp}.*

Furthermore, we can state that only those items of an export part can be deleted which are not used in some import.

Proposition 3.7 *In all EGO's o a subgraph U of $State(exp(o))$ can only be deleted if there is no other EGO o' and a use relation u between them such that U is in the image of $f_{in(k)}$.*

None of the EGO's imports an element more than once from different EGO's.

Proposition 3.8 *Each element in the import part $State(imp(o))$ of an EGO o has a unique preimage in the disjoint union of all connected interface state graphs $State(int(k))$ for all k with $out(k) : int(k) \to imp(o)$.*

4 Summary and Open Problems

DIEGO module systems are a first attempt to combine the advantages of distributed graph transformation systems with the advantages of encapsulated hierarchical graphs and graph types. They provide a *graph grammar module concept* which supports distributed programming and information hiding. The basic components of a DIEGO module system are so-called EGO modules which are connected to each other by means of use relations. We have to emphasize that the concept of DIEGO module systems is more specific than that of modules in Modula-2 or packages in Ada. EGO modules do not export an arbitrary number of types and procedures at their export interface but just a single graph type with its accompanying productions (and the initial state which is needed within the implicitly defined create operations). Therefore, DIEGO module concepts are very similar to those of abstract data type modules as defined in [4]. There seems to be also a relationship to the class concept of languages like C++ or Eiffel except of inheritance concepts. Future research is still necessary in order to

- allow more advanced graph transformation systems (with other transformation approaches, application conditions, transactions, etc.)
- incorporate the notion of graph schemata from hierarchical graph types into the new formalism,
- study other interconnection concepts for modules, like inheritance, where heirs either have full access to the bodies of their ancestors or get only restricted access via special inheritance interfaces, and
- extend the HD-graph model with a global network layer and a local graph layer to a true hierarchical data model with an arbitrary number of layers.

References

[1] H. Ehrig and G. Engels. Towards a Module Concept for Graph Transformation Systems. Technical Report 93-34, Leiden University (The Netherlands), 1993.

[2] G. Engels and A. Schürr. Hierarchical Graphs, Graph Types and Meta Types. Extended Abstract submitted to SEGRAGRA'95.

[3] H.-J. Kreowski and S. Kuske. On the Interleaving Semantics of Transformation Units - A Step into GRACE. Submitted to proceedings of Graph Grammar Workshop in Williamsburg, 1994.

[4] M. Nagl. Programming in the Large (in German). Springer Verlag, 1990.

[5] A. Schürr. PROGRESS–Editor: A Text–Oriented Hybrid Editor for PROgrammed Graph REwriting SyStems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*, page 67. Springer, 1991. Lecture Notes in Computer Science 532.

[6] G. Taentzer. Hierarchically Distributed Graph Transformation. Submitted for proceedings of Graph Grammar Workshop, Williamsburg, 1994.