URL: http://www.elsevier.nl/locate/entcs/volume51.html 14 pages

An Abstract Module Concept for Graph Transformation Systems [1]

Marta Simeoni [2]

Dipartimento di Informatica, Università 'Cà Foscari' di Venezia

Abstract

Graph transformation systems are a well known formal specification technique that support the rule based specification of the dynamic behaviour of systems. Recently, many specification languages for graph transformation systems have been developed, and modularization techniques are then needed in order to deal with large and complex graph transformation specifications, to enhance the reuse of specifications, and to hide implementation details. In this paper we present an abstract categorical approach to modularization of graph transformation systems. Modules are called *cat*–modules and defined over a generic category *cat* of graph transformation specifications and morphisms. We describe the main characteristics and properties of *cat*–modules, their interconnection operations, namely union, composition and refinement of modules, and some compatibility properties between such operations.

1 Introduction

Graph grammars and graph transformation systems are well known rule based mechanisms for the manipulation of graphs and graphical structures. Graph grammars usually specify graph languages, while graph transformation systems typically describe dynamically evolving systems where graphs are states and graph transformations are state transitions.

In recent years many specification languages for graph grammars and graph transformation systems have been developed. For example the PROgramming with Graph REwriting Systems – PROGRES – (see [17]) and the Algebraic Graph Grammar (AGG) system (see [7]). Modularization techniques are thus needed for dealing with large and complex graph transformation specifications, to enhance the reuse of specifications and to hide implementation details.

[1] This work has been supported by the EEC TMR network GETGRATS (General Theory of Graph Transformation Systems). The complete investigation on this subject is the result of a collaboration with Martin Große–Rhode, Dirk Janssens and Francesco Parisi Presicce

[2] Email:simeoni@dsi.unive.it

©2002 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.

There are already various proposals about this topic. The first one has been introduced in [3], but more worked out approaches are the PROGRES packages (see [16]), the GRACE graph transformation units and modules (see [10,11]) and the DIEGO modules (see [18]). All these three approaches to modularization of graph transformation systems fullfil the basic requirements for modules, that means, they support implementation hiding and *use relation* for modules (i.e. the possibility to specify how a module can use the features exported by another module).

What is missing in the PROGRES, GRACE and DIEGO approaches is a formal abstract theory on modularization, with concrete results concerning both the compatibility properties between the modules interconnection operations, and the preservation of behaviour of the module components. The latter is a main requirement to be satisfied by graph transformation specifications, since they describe the dynamical behaviour of systems. For modules, it means that the semantical requirements specified in the export (resp. import) interface, have to be preserved in the body component.

In this paper we present an abstract categorical approach to modularization of graph transformation systems, which has been introduced in [15] and instantiated for both Local Action Systems (see [9]) and typed graph transformation systems over the Double Pushout approach (see [2]).

The abstract module concept is inspired by the algebraic specification modules introduced in [6]. Modules are defined over a generic category *cat* of graph transformation specifications and morphisms, and they are called *cat*–modules.

A *cat*–module has three components: an import interface, an export interface and a body. The interfaces describe the relations to other *cat*–modules. More precisely, the export interface specifies the features offered by the module to other modules or the environment, the import interface specifies the parts that are required from other modules, and can be used in the body to implement the features to be exported.

Each component is a graph transformation system specification which consists of a set of graph transformation rules that define its basic steps. The interconnections between the module components are modeled by morphisms of graph transformation specifications.

Guided by the analogous interconnection operations for algebraic specification modules, we define the union, composition and refinement of modules. The composition realizes a use relation between modules, while the refinement realizes a kind of inheritance relation. We prove compatibility properties between the proposed operations: an important one is the compatibility of composition with respect to refinement of modules.

cat–modules are introduced and formalized independently of any particular approach on graph transformation and by pointing out all the categorical requirements to be fulfilled by any instantiation of *cat*–modules over a concrete framework. The advantage of this abstract formalization is that we obtain a general approach to modularization with concrete but still general results, like the definition and the compatibility properties of the interconnection operations for *cat*–modules. Moreover, the instantiation of *cat*–modules over a concrete approach follows automatically by providing formal proofs to the generic categorical requirements stated for *cat*–modules.

The paper is organized as follows. The main characteristics and properties of *cat*–modules are described and formalized in Section 2, while the interconnection operations for *cat*–modules, and their compatibility properties are presented in Section 3. Section 4 provides some concluding remarks.

2 *cat*–modules

In this section we introduce an abstract notion of module for graph transformation systems, inspired by the algebraic specification modules (see [6]).

Graph transformation systems are specified by a set of rule names and a mapping associating to each name the underlying graph transformation rule. More precisely we consider specifications having rules typed over a specific *type system*, modeled, for instance, by an alphabet for node and/or edge labels, or a type graph in the sense of [1]. Since we are interested in defining modules in a categorical setting, we suppose that a generic category *cat* of graph transformation specifications and morphisms can be defined. Hence, we define the abstract modules over *cat*, and call them *cat*–modules. In what follows we describe their main characteristics and properties, and point out the categorical requirements to be fulfilled by any instantiation of *cat*–modules over a concrete framework.

A *cat*–module has three components: an import interface, an export interface and a body. Each of the three components is a graph transformation specification. Hence, rules are not only present in the body component, but also in the interfaces and this means that, unlike the programming language modules, beside the names of the imported and exported features it is possible to specify also their behaviour.

We proceed now with a stepwise description of the role and meaning of the *cat*–module components and their interconnections.

Import interface

The import interface specifies the features provided by other modules and used in the body component to implement new features.

Modules with an empty import interface are the basic building blocks in the bottom-up development of a modular system. Beside the bottom-up technique, however, *cat*–modules allow also a top–down development of modular systems. In fact, the features to be imported from other modules are specified not only by their names but also by rules describing their behaviour. This

allows their concrete implementation by other modules to be postponed, and realizes a mechanism supporting generic imports.
Furthermore, since the concrete modules which provide the imported features are not explicitly referenced, any module realizing the same behaviour can be used. An advantage of this importing mechanism is that it makes a selective import of features possible. In general, this is not allowed in the case of programming language modules, where all the exported features of the modules specified in the import interface have to be imported.

Export interface

The export interface specifies the features realized by the module and offered to the external environment. Again, the exported features are not only specified by their names, but also by rules describing their behaviour: in this case, the rules specify the pre- and postconditions of their implementation.
The export interface is the only visible component of the module from the outside environment, i.e the export rules typed over the export type system are the only accessible resources. This means that the implementation details of the exported rules are hidden inside the module.

Body

The body of a *cat*–module contains all the local and imported rules needed to implement the exported features. The idea for supporting implementation, is that each exported feature is realized by a suitable combination of rules of the body.
The type system of the export interface is intended to be a subtype of the body type system: in this way the body can use private types not visible from the external environment, i.e. data hiding is supported. Also the type system of the import interface is usually a subtype of the body type system: in this way the local rules can use additional types w.r.t the imported ones.

The three components of a *cat*–module have to be related in such a way that both the imported features are embedded into the body (so that the body can use them) and the exported features are implemented using rules of the body (i.e. local rules and imported ones). That means, two kinds of relations have to be established: one between the export interface and the body and another one between the import interface and the body. Both of them have to relate first the type systems of the two components and then their rules.

Import–body relation

The task of the import–body relation in a *cat*–module is to include the imported rules into the body, so that the body can use them for implementing the exported rules: we model it by an injective *plain morphism*.

A plain morphism relates the source type system with the target one and establishes a one–to–one correspondence between rules: it associates to each rule of the source specification a single rule of the target one, in such a way that the translation of the source rule over the target type system yields exactly the associated target rule. We require the following property for plain morphisms:

> Requirement: Specifications and plain morphisms define a category which is closed under pushouts.

Export–body relation

The formalization of the export–body relation is quite involved because each rule of the export interface has to be associated with its implementation via rules of the body: it is modeled by a *refinement morphism*.
We use refinements to support the implementation task. Refinements are the basic steps in the development of complex system specifications. Starting from an abstract description of the system's behaviour, stepwise refinements yield more and more concrete specifications, that should finally be directly implementable on a machine.
A refinement of a more abstract specification by a more concrete one is given by associating with each rule of the more abstract specification a combination of rules of the more concrete specification, in such a way that the composed rule (i.e. the rule resulting from the combination) coincides with the translation of the abstract rule over the finer type system.
As explained before, the export type system is intended to be a subtype of the body type system: by modeling the export–body relation through refinements we realize both data and implementation hiding because both the internal types and the internal steps of the body are not visible by the module users (they can only access the module via the export interface).
Any plain morphism can be embedded into a refinement morphism associating each abstract rule with just one concrete rule. The first requirement for specifications and refinement morphisms is the following one:

> Requirement: Specifications and refinement morphisms define a category which has, as a subcategory, the category of specifications and plain morphisms.

Moreover, we require the following *pushout with inclusions property* for refinement morphisms, needed for defining the composition operation between *cat*–modules in the next section.

> Requirement: The category of specifications and refinement morphisms is closed under pushouts, if one of the morphisms is an injective plain morphism. The induced morphisms are again a refinement and an injective plain morphism (i.e the injective plain morphism is preserved).

Having described the components of a *cat*–module and their interconnections, we can concretely formalize a *cat*–module as follows.

Definition 2.1 [*cat*–Module] A *cat–module* $MOD = (IMP \overset{m}{\rightarrow} BOD \overset{r}{\leftarrow} EXP)$ is given by specifications IMP, BOD, and EXP, an injective plain morphism $m : IMP \rightarrow BOD$, and a refinement $r : EXP \rightarrow BOD$. It can be visualized by:

$$\begin{tikzcd} & EXP \arrow[d, "r"] \\ IMP \arrow[r, "m"] & BOD \end{tikzcd}$$

Fig. 1 shows an example of a module exporting two rules p and u which are implemented (refined) in the body component by a suitable combinination of rules w, q, s and s, k, r, respectively. The idea is that the refinement morphism specifies how the body rules have to be combined in order to implement the corresponding rules of the export interface. The module imports a rule t, which is included into the body component via a plain morphism mapping t to q.

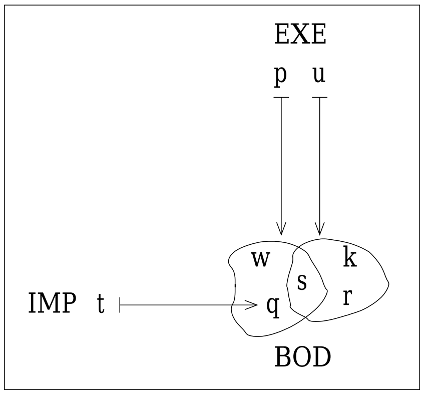

Fig. 1. A module

Since plain morphisms are just particular cases of refinement morphisms, both the components and the morphisms of a *cat*–module can be modeled in the category of specifications and refinement morphisms. Hence, the generic category *cat* is exactly the category of specifications and refinement morphisms.

Semantics

The semantics of a *cat*–module is given by the semantics of its three components, related by the mappings induced by the specification morphisms. The existence of such induced mappings, however, have to be explicitely required.

REQUIREMENT: Specification morphisms induce mappings on the semantics.

Since the export interface is the only component of a *cat*–module visible from the outside environment, it follows that only the export interface semantics is visible.

We have already pointed out that *cat*–modules allow a generic import of the features realized by other modules, and used for implementing the exported features. Generic import makes it possible to consider each module as a self contained entity with fully defined semantics. Hence, the semantics of a modular system can be defined from the semantics of its module components. This kind of compositional semantics is not allowed in programming language modules, where the semantics of a module is only defined if all modules required at the import interface are actually imported.

Preservation of behaviour property

The main property we require to graph transformation specifications and morphisms, in order to suitably model the dynamical behaviour of systems is the following *preservation of behaviour property*:

> REQUIREMENT: The dynamical behaviour of a graph transformation specification has to be preserved along plain and refinement morphisms.

This property is particularly important for the export–body morphism of *cat*–modules: since the export interface is the only component of the module visible from the outside environment, the preservation of the export semantics along the refinement defines a correctness criterion for the implemented features.

We conclude this section by describing the basic ways to relate *cat*–modules, which are the basic for defining the interconnection operations, in the next section.
cat–modules can be related via *cat–module morphisms*, consisting of a triple of plain morphisms, compatible with their internal module connections.

Definition 2.2 [*cat*–module morphism] Let $MOD = (IMP \overset{m}{\to} BOD \overset{r}{\leftarrow} EXP)$ and $MOD' = (IMP' \overset{m'}{\to} BOD' \overset{r'}{\leftarrow} EXP')$ be *cat*–modules. A *cat*–module morphism $mod : MOD \to MOD'$ is a triple $(mod_I : IMP \to IMP', mod_B : BOD \to BOD', mod_E : EXP \to EXP')$ of plain morphisms such that the following are commuting diagrams in *cat*.

$$\begin{tikzcd}
IMP \arrow[r, "m"] \arrow[d, "mod_I"'] \arrow[dr, phantom, "="] & BOD \arrow[d, "mod_B"] \\
IMP' \arrow[r, "m'"'] & BOD'
\end{tikzcd}
\qquad
\begin{tikzcd}
BOD \arrow[d, "mod_B"'] \arrow[dr, phantom, "="] & EXP \arrow[l, "r"'] \arrow[d, "mod_E"] \\
BOD' & EXP' \arrow[l, "r'"]
\end{tikzcd}$$

Having defined *cat*–modules and *cat*–modules morphisms, we can prove the following proposition:

Proposition 2.3 *[15]* cat*–modules and* cat*–module morphisms form a category called* **MOD**.

Another useful relation between *cat*–modules is the submodule relation: a *cat*–module *MOD* is a submodule of another module *MOD′* if there exists a *cat*–module morphism $mod = (mod_I, mod_B, mod_E) : MOD \to MOD'$ such that the plain morphism mod_B between the body components keeps imported rules and local ones separated: this condition ensures that the rules in the two components have the same role.

Definition 2.4 [Submodule] Let $MOD = (IMP \overset{m}{\to} BOD \overset{r}{\leftarrow} EXP)$ and $MOD' = (IMP' \overset{m'}{\to} BOD' \overset{r'}{\leftarrow} EXP')$ be two *cat*–modules. MOD is a *submodule* of MOD' if there exists a *cat*–module morphism $mod = (mod_I, mod_B, mod_E) : MOD \to MOD'$ such that $mod_B : BOD \to BOD'$ maps imported rules into imported rules and local rules into local rules. The morphism mod is called a *submodule morphism.*

We use the submodule relation for defining the union of *cat*–modules, in the next section. However, in order to be able to define such operation, the following property have to be required:

> REQUIREMENT: Pushouts in the category of specifications and plain morphisms are preserved in the category of specifications and refinement morphisms.

Having this property, we can prove that the category **MOD** has pushouts w.r.t the submodule relation (which is the basis for defining the union operation).

Proposition 2.5 *[15]* **MOD** *has pushouts w.r.t submodules.*

In this section we have described the components and interconnections of *cat*–modules, their semantics, and their basic properties. Moreover, we have introduced the basic relations between *cat*–modules. On this basis, we define in the next section some possible interconnection operations for *cat*–modules.

3 An Algebra of *cat*–modules

Modules have been introduced as structuring means for the development of complex specifications. Hence, beside the definition of a module concept for graph transformation systems it is important to define also interconnection operations for modules.
Guided by the operations for algebraic specification modules (see [6]), we introduce analogous operations for *cat*–modules, namely *union*, *composition* and *module refinement.* Due to the categorical approach, the connections between modules are modeled by morphisms and the results of operations are specified abstractly by their universal properties.

3.1 Union of cat*–modules*

The union of two *cat*–modules MOD_1 and MOD_2 is defined w.r.t. an explicitly defined common submodule MOD_0 , where the submodule relationship has been introduced in definition 2.4. This allows to control explicitly which items of MOD_1 and MOD_2 are to be considered as shared and which ones as local, independently of the names chosen locally. In this way items from MOD_1 and MOD_2 are identified in the union if and only if they are images of one corresponding item in MOD_0, whose embedding into MOD_1 and MOD_2 makes the identifications explicit. More concretely, the union $MOD_1 \oplus_{MOD_0} MOD_2$ can be obtained by taking first the disjoint union of the corresponding components of MOD_1 and MOD_2 and then identifying the parts in common contained in MOD_0.

Definition 3.1 Let MOD_1 and MOD_2 be two *cat*–modules and let MOD_0 be a submodule of both MOD_1 and MOD_2 via the submodule morphisms $mod_1 : MOD_0 \to MOD_1$ and $mod_2 : MOD_0 \to MOD_2$. The *union* $MOD_3 = MOD_1 \oplus_{MOD_0} MOD_2$ of MOD_1 and MOD_2 w.r.t. MOD_0, mod_1 and mod_2, is given by a pushout of MOD_0, mod_1 and mod_2 in **MOD**.

$$\begin{tikzcd}
 & MOD_0 \arrow[dl] \arrow[dr] & \\
MOD_1 \arrow[dr, dashed] & p.o & MOD_2 \arrow[dl, dashed] \\
 & MOD_3 &
\end{tikzcd}$$

The existence of pushouts in **MOD** w.r.t submodules is stated in proposition 2.5.
The union operation is specially useful when it is used in combination with the composition operation described below: in that case, it allows the importing of features realized by different modules.
Finally, the following facts can be proved using only standard properties of the involved categories. For their complete proofs we refer to [12] where they are proved for algebraic specification modules.

Proposition 3.2 (i) *Compatibility of union and submodule relation: MOD_1 and MOD_2 are submodules of $MOD_1 \oplus_{MOD_0} MOD_2$ (it follows from proposition 2.5 and standard properties of pushouts);*

(ii) *Associativity of union: if MOD_0 is a submodule of MOD_1 and MOD_2, and MOD_3 is a submodule of MOD_2 and MOD_4, then $(MOD_1 \oplus_{MOD_0} MOD_2) \oplus_{MOD_3} MOD_4 \cong MOD_1 \oplus_{MOD_0} (MOD_2 \oplus_{MOD_3} MOD_4)$.*

3.2 Composition of cat*–modules*

The composition operation realizes a *use relation* for *cat*–modules: a module $MOD = (IMP \overset{m}{\to} BOD \overset{r}{\leftarrow} EXP)$ can *use* another module $MOD' = (IMP' \overset{m}{\to} BOD' \overset{r}{\leftarrow} EXP')$ if the import interface of the first one can be related with the export interface of the second one via a plain morphism, called *interface*

morphism, between the import interface of MOD and the export interface of MOD', $h : IMP \to EXP'$.
The new module resulting from the composition of MOD and MOD' has the import interface of MOD', the export interface of MOD and a body implementing the features of both MOD and MOD'. The basic categorical requirement for defining this operation is the *pushout with inclusions* property for the category *cat*, which allows the construction of the composed body.

Definition 3.3 Let $MOD = (IMP \overset{m}{\to} BOD \overset{r}{\leftarrow} EXP)$ and $MOD' = (IMP' \overset{m'}{\to} BOD' \overset{r'}{\leftarrow} EXP')$ be two *cat*–modules and let $h : IMP \to EXP'$ be a plain morphism, called the interface morphism. The composition $MOD'' = MOD' \circ_h MOD$ is defined by $MOD'' = (IMP'' \xrightarrow{m^* \circ m'} BOD'' \xleftarrow{r^* \circ r} EXP'')$ where: $IMP'' = IMP'$, $EXP'' = EXP$ and BOD'' is a pushout object (and m^* and r^* are the induced morphisms) of the following diagram in *cat*.

$$\begin{tikzcd}
 & & EXP \arrow[d, "r"] \\
 & IMP \arrow[r, "m"] \arrow[d, "h"] & BOD \arrow[dd, dashed, "r^*"] \\
 & EXP' \arrow[d, "r'"] \arrow[r, phantom, "p.o."] & \\
IMP' \arrow[r, "m'"] & BOD' \arrow[r, dashed, "m^*"] & BOD''
\end{tikzcd}$$

Note that the interface morphism h can be just the identity morphism $h = id$, i.e. the composition operation is always defined. In fact, we can recover the $h \neq id$ case by using the intermediate *cat*–module $(EXP' \xrightarrow{id} EXP' \xleftarrow{h} IMP)$ and composing it with the given modules.
The composition and union operations can be used in combination, in order to realize the importing of features from different modules: if a module MOD needs to import features realized by two different modules MOD_1 and MOD_2, it is sufficient to perform first the union of MOD_1 and MOD_2, and then the composition of the resulting module with MOD.
As for the union operation, the following basic properties of composition can be deduced immediately from their universal properties, as shown explicitly in [6] for algebraic specification modules.

Proposition 3.4 (i) *Associativity of composition:* $(MOD'' \circ_{h_1} MOD') \circ_{h_2} MOD \cong MOD'' \circ_{h_1} (MOD' \circ_{h_2} MOD)$ *due to standard properties of com-*

position of pushouts.

$$\begin{tikzcd}
 & & & & EXP \arrow[d, "r"] \\
 & & & IMP \arrow[r, "m"] \arrow[d, "h_1"'] & BOD \arrow[d, dashed] \arrow[dd, dotted, bend left] \\
 & & & EXP' \arrow[d, "r'"'] & \\
 & & IMP' \arrow[r, "m'"] \arrow[d, "h_2"'] & BOD' \arrow[r, dashed] \arrow[dd, dotted] & {} \arrow[dd, dashed] \\
 & & EXP'' \arrow[d, "r''"'] & & \\
 & IMP'' \arrow[r, "m''"] & BOD'' \arrow[r, dotted] \arrow[rr, dashed, bend right] & {} \arrow[r, dotted] & BOD'''
\end{tikzcd}$$

(ii) *Compatibility of composition and submodule: if* MOD_1 *is a submodule of* MOD_3 *and* MOD_2 *is a submodule of* MOD_4 *then* $(MOD_1 \circ_{h_1} MOD_2)$ *is a submodule of* $(MOD_3 \circ_{h_3} MOD_4)$*, with* h_1 *and* h_3 *interface morphisms form* MOD_1 *to* MOD_2 *and from* MOD_3 *to* MOD_4*, respectively.*

(iii) *Symmetric Distributivity of union and composition: let* MOD_0 *be a submodule of* MOD_1*,* MOD_2 *and* MOD'_0 *a submodule of* MOD'_1*,* MOD'_2*. Let* $h_i : MOD_i \to MOD'_i$*,* $i = 0, 1, 2$ *be interface morphisms such that* h_0 *is the restriction of* h_1 *and* h_2*. Then*

$$(MOD_1 \oplus_{MOD_0} MOD_2) \circ_{h_1 \oplus_{h_0} h_2} (MOD'_1 \oplus_{MOD'_0} MOD'_2) \cong$$
$$(MOD_1 \circ_{h_1} MOD'_1) \oplus_{MOD_0 \circ_{h_0} MOD'_0} (MOD_2 \circ_{h_2} MOD'_2).$$

3.3 Refinement of cat*–modules*

The refinement relation between single specifications can be extended componentwise to *cat*–modules. We say that a *cat*–module is refined by another one if there exist three refinement morphisms between their corresponding components that are compatible with the internal module connections.

There are no special requirements for defining this operation: it realizes a sort of inheritance relation for *cat*–modules.

Definition 3.5 Let $MOD = (IMP \overset{m}{\to} BOD \overset{r}{\leftarrow} EXP)$ and $MOD' = (IMP' \overset{m'}{\to} BOD' \overset{r'}{\leftarrow} EXP')$ be two *cat*–modules. A *cat–module refinement* $r_{MOD} = (r_I, r_B, r_E) : MOD \to MOD'$ is a triple of refinements $r_I : IMP \to IMP'$, $r_B : BOD \to BOD'$ and $r_E : IMP \to EXP'$ such that the following diagrams commute in *cat.*

$$\begin{tikzcd}
IMP \arrow[r, "m"] \arrow[d, "r_I"] \arrow[dr, phantom, "="] & BOD \arrow[d, "r_B"] \arrow[dr, phantom, "="] & EXP \arrow[l, "r'"'] \arrow[d, "r_E"] \\
IMP' \arrow[r, "m'"] & BOD' & EXP' \arrow[l, "r'"']
\end{tikzcd}$$

Note that *cat*–module morphisms as defined in 2.2 are particular cases of *cat*–module refinements.

Note moreover that each component of the refined module can contain "local rules" which do not depend from the rules of the more abstract module, i.e the refined module can be specialized with features not visible from the abstract one.

We conclude this section by showing two compatibility properties of *cat*–module refinements w.r.t the union and composition of *cat*–modules. To this end, analogously to the category **MOD** we define the category **MOD-Ref** of *cat*–modules and *cat*–module refinements and prove that it is closed under pushouts, if one of the involved morphisms is just an injective *cat*–module morphism (which follows directly from the pushout with inclusion property).

Proposition 3.6 *[15] Any pair* $(r_{MOD} : MOD_0 \to MOD_1, mod : MOD_0 \to MOD_2)$ *consisting of a* cat*–module refinement* r_{MOD} *and an injective* cat*–module morphism* mod *has pushouts in* **MOD-Ref**.

The compatibility properties of *cat*–module refinements w.r.t. the union or composition of *cat*–modules are based on the following compatibility condition:

Definition 3.7 Let $m : MOD_0 \to MOD_1$ be a submodule morphism and $r_{mod_0} : MOD_0 \to MOD'_0$ be a *cat*–module refinement. A *cat*–module refinement $r_{mod_1} : MOD_1 \to MOD'_1$ is *compatible* with m and r_{mod_1} if there exists a submodule morphism $m' : MOD'_0 \to MOD'_1$ such that the following diagram commutes:

$$\begin{tikzcd}
MOD_0 \arrow[r, "m"] \arrow[d, "r_{mod_0}"'] & MOD_1 \arrow[d, "r_{mod_1}"] \\
MOD'_0 \arrow[r, "m'"'] & MOD'_1
\end{tikzcd}$$

Having this definition, we can prove the following properties:

Proposition 3.8 *[15] The union of* cat*–module refinements is the cat–module refinement of the union. The composition of* cat*–module refinements is the* cat*–module refinement of the composition.*

4 Concluding remarks

In this paper we have presented an abstract categorical approach to modularization of graph transformation systems. Two concrete instantiations of this approach have been investigated in [15]: the first one for typed graph transformation systems over the double pushout approach, and the second one for local action systems.

cat–modules are inspired by the algebraic specification modules. There is, however, a main conceptual difference between the two specification formalisms:

graph transformation system specifications are used for formally specifying dynamically evolving systems. Hence, it is important to preserve the dynamical behaviours of systems. Such dynamical aspect is not present in algebraic specifications since each of them specifies an algebra (initial semantics) or a class of algebras (loose semantics), that model only the static functional view of the components.

The relationships and differences between *cat*–modules and other modularization approaches to graph transformation need also to be discussed. However, due to space limitation, we refer to [15] for a detailed presentation of the main approaches and their comparison w.r.t. *cat*–modules.

The ideas for future work are concerned with both the development of other instantiations of *cat*–modules, and the extension of the refinement relation. In the first case we are interested in intantiating *cat*–modules for any formalism satisfying the required properties (for example graph transformations in the Single Pushout approach, Algebra Rewriting, Petri nets). In the second case, we are interested in extending the refinement relation for modeling case distinctions, like *if–then–else* and *case* constructs, and iteration. A first attempt in this direction can be found in [8], where some extensions of refinements are discussed for modules of typed graph transformation systems over the double pushout approach.

References

[1] A. Corradini, U. Montanari, and F. Rossi "Graph processes" *Special issue of Fundamenta Informaticae*, Vol. 26(3,4) pp. 241–266 (1996)

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe "Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach" In [14], pp. 163–246

[3] H. Ehrig, G. Engels "Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems" *Proc. 5th International Workshop on Graph Grammars and their application to Computer Science, Williamsburg'94, LNCS 1073*, pp. 137–154 (1996)

[4] H. Ehrig, G. Engels H.-J. Kreowski, G. Rozenberg, ed. "Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools" World Scientific (1999)

[5] H. Ehrig, H.J. Kreowsky, U. Montanari, G. Rozenberg, ed. "Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism, and Distribution" World Scientific (1999)

[6] H. Ehrig, B. Mahr "Fundamentals of Algebraic Specifications 2: Module Specifications and Constraints" *EATCS Monographs on Theoretical Computer Science* Vol. 21 Springer Verlag, Berlin (1990)

[7] C. Ermel, M. Rudolf, G. Taentzer "The AGG approach: Language and Environment" in [4], pp. 487–546 (1999)

[8] M. Große–Rhode, F. Parisi Presicce, M. Simeoni "Formal Software Specification with Refinements and Modules of Typed Graph Transformation Systems" *Journal of Computer and System Sciences*, to appear.

[9] D. Janssens "Actor Grammars and Local Actions" In [5], pp.57–106 (1999)

[10] H-J. Kreowski, S. Kuske "On the Interleaving Semantics of Transformation Units – A Step into GRACE", *LNCS 1073*, pp. 89–106 (1996)

[11] H-J. Kreowski, S. Kuske "Graph Transformation Units and Modules" In [4], pp. 607–638

[12] F. Parisi Presicce "Inner and Mutual Compatibility of operations on module specifications" *Technical Report 86-06 TU Berlin* (1986)

[13] F. Parisi Presicce "Transformation of Graph Grammars", *LNCS 1073*, pp. 428–442 (1996)

[14] G. Rozenberg, ed. "Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations" World Scientific (1998)

[15] M. Simeoni "A Categorical Approach to Modularization of Graph Transformation Systems using Refinements" *PhD Thesis, Universitá La Sapienza di Roma* (2000)

[16] A. Schürr, A. Winter "UML Packages for PROgrammed Graph REwriting Systems" *Proc. 6th international Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, LNCS 1764*, pp. 396–409 (1999)

[17] A. Schürr, A. Winter, A. Zündorf "The PROGRES approach: Language and Environment" in [4], pp. 547–668 (1999)

[18] G. Taentzer, A. Schürr "DIEGO, another step towards a module concept for graph transformation systems" *Proc. Graph Rewriting and Communication, SEGRAGRA'95* Electronic Notes of TCS, Vol. 2, (1995) http://www.elsevier.nl/locate/entcs/volume2.html