Science of Computer Programming 107–108 (2015) 2–18

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Algebraic graph transformations with inheritance and abstraction

Michael Löwe, Harald König, Christoph Schulz *, Marius Schultchen

FHDW Hannover University of Applied Sciences, Freundallee 15, 30173 Hannover, Germany

ARTICLE INFO

Article history:
Received 4 May 2014
Received in revised form 7 February 2015
Accepted 10 February 2015
Available online 24 February 2015

Keywords:
Graph transformation
Inheritance
Abstraction
Adhesive HLR category

ABSTRACT

In this paper, we propose a new approach to inheritance and abstraction in the context of algebraic graph transformation by providing a suitable categorial framework which reflects the semantics of class-based inheritance in software engineering. Inheritance is modelled by a type graph T that comes equipped with a partial order. Typed graphs are arrows with codomain T which preserve graph structures up to inheritance. Morphisms between typed graphs are "down typing" graph morphisms: An object of class t can be mapped to an object of a subclass of t. Abstract classes are modelled by a subset of vertices of the type graph. We prove that this structure is an adhesive HLR category, i.e. pushouts along extremal monomorphisms are "well-behaved". This infers validity of classical results such as the Local Church–Rosser Theorem, the Parallelism Theorem, and the Concurrency Theorem.

1. Introduction and related work

1.1. Modelling object-oriented systems

Developing appropriate models to mimic reality has always been an important part of software engineering. However, the relation between coding and modelling has changed over time. Today, model-driven engineering focuses on generating code from appropriately detailed and formalised models, hoping that developing the model and using a mature and well-tested code generator is less error-prone than letting programmers write most of the code themselves. This reasoning, however, is only valid if model development is relatively easy. Typically, different graphical notations help people to structure the problem in various ways. Consequently, *graphs* or graph structures play an important role in software engineering today, compare e.g. the UML [1], a language which is currently the de facto standard for modelling object-oriented systems.

If one looks more closely at object-oriented systems, which consist of a type level with classes, associations, etc. accessible at design time and an instance level with objects, links, etc. at run-time, one realises that it is impossible to analyse or build object-oriented software in an efficient way without making use of specialisation or *inheritance*.[1] Inheritance allows us to factor out common data and behaviour into separate classes which can be documented and tested separately. By differentiating between classes providing an interface (only) and classes implementing that interface, a separation between

* Corresponding author.
E-mail addresses: Michael.Loewe@fhdw.de (M. Löwe), Harald.Koenig@fhdw.de (H. König), Christoph.Schulz@fhdw.de (C. Schulz), Marius.Schultchen@web.de (M. Schultchen).

[1] In this paper, we do not differentiate between interface inheritance (specialisation or subtyping) and implementation inheritance (class inheritance of subclassing), because the differences are mostly relevant in the context of type theory, which we do not discuss, and because most mainstream OOP languages do not differentiate between these concepts; even in Java, subclassing always implies subtyping.

http://dx.doi.org/10.1016/j.scico.2015.02.004
0167-6423/

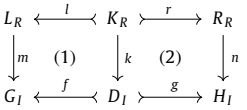

Fig. 1. Double-pushout transformation according to [4].

interface and implementation can be made explicit. Moreover, it allows to extend existing behaviour in a non-intruding way, which is sometimes called the "Open–Closed Principle". When inheritance is applied top–down, it enables the use of subtype polymorphism and makes parallel development against common interfaces possible. When applied bottom-up, it permits refactoring and the cleanup of interfaces and code. All in all, inheritance is one of the most important features of the object-oriented paradigm [2,3].

In addition, the notion of *abstraction* allows to define abstract data types (ADT), which have to be specialised for concrete use cases. It is a natural consequence of the inheritance feature to be able to make the difference between abstract and concrete classes. Abstract classes are "unfinished" and need to be completed by subclasses. So no instances of abstract classes are desirable as such objects would not be able to understand and react to the complete set of messages their interface allows. In contrast, concrete classes guarantee that their instances are fully functional. In an object-oriented system, all objects are instances of concrete classes. When transforming such systems (see below), it is highly desirable that this property be preserved.

It should also be stressed that (interface) inheritance typically requires *multiple inheritance*, i.e., that a type not only inherits from a single type but from a whole set of other types. This enables designers to formulate the key idea that a class implements behaviour specified by many types (interface inheritance) or extends the behaviour and/or state implemented by many classes (implementation inheritance). If multiple inheritance were not supported, the designer would be forced to create fat interfaces[2] or to introduce complicated delegation patterns, which considerably defeats the goal of building an object-oriented system from small and coherent parts.

It follows that it is sensible to require that the graphical notation directly support aspects of inheritance and abstraction, without being forced to transform them to lower-level constructs. Keeping the models at a high level of abstraction helps in understanding what they are about and how they change. Lowering the abstraction level of the model necessarily leads to constructs that are more difficult to read and understand. Additionally, if such a lower-level model is transformed (see below), the resulting model somehow has to be translated back to a high-level representation, which may be difficult or even impossible in some cases.[3]

1.2. Graph transformations

On the one side, graphs are well suited for modelling static aspects of software, e.g. the class and inheritance structure. On the other side, behavioural aspects of the system, e.g. state changes, can be modelled using *graph transformations* which formally describe when and how a graph (here: state of an object-oriented system) can change into another graph (here: another system state). Typically, such a transformation is modelled either by a graph transformation rule $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ with source graph L, target graph R, and total graph morphisms l and r, as it is done in the double-pushout approach (DPO, [4]), or as a single partial graph morphism $L \overset{r}{\rightarrow} R$, as it is done in the single-pushout approach (SPO, [5]). In Fig. 1 you can see an example of the double-pushout approach where the graph transformation rule is defined at the top and the graph to be transformed is found at the bottom; a matching relation then defines which parts of the graph the rule shall apply to. Both squares are so-called *pushout* squares. In the rule, l specifies which graph elements are to be deleted (namely those which are not in the image of l), whereas r determines which graph elements are to be added (namely those which are not in the image of r). When G_I gets transformed along l and r, the effects of l and r on L_R are *mirrored* on G_I, namely on the part of G_I which is reached by m. So f and g do the same on $m(L_R)$ in the context of G_I as l and r do on L_R.

In this paper, we concentrate on the double-pushout approach as depicted in Fig. 1. To be able to apply the double-pushout approach to a system that does not consist of standard graphs and standard graph homomorphisms only, as e.g. typed graphs, we need to ensure that the category we define is an *adhesive HLR category*. Adhesive high-level replacement (HLR) categories as introduced in [6,4] combine high-level replacement systems [7] with the notion of adhesive categories [8] in order to be able to generalise the double-pushout transformation approach from graphs to other high-level structures, as e.g. attributed graphs [9] and Petri nets using a categorial framework. Generally, adhesiveness abstracts from exactness properties like compatibility of union and intersection of sets. Due to special properties that guarantee the preservation of pushouts and pullbacks in certain situations, many useful results of the theory of graph transformation can be obtained, for example the Local Church–Rosser Theorem for pairwise analysis of sequential and parallel independence [4, Theorem 5.12], the Parallelism Theorem for applying independent rules and transformations in parallel [4, Theorem 5.18], or the Concurrency Theorem for applying E-related dependent rules simultaneously [4, Theorem 5.23]. Algebraic graph trans-

[2] A fat interface is an interface with many operations.

[3] The second case may happen if the transformation does not preserve some of the properties required by the higher-level models.