DIAGRAMATIC SOFTWARE SPECIFICATION

ADRIAN RUTLE[1], YNGVE LAMO[1] AND UWE WOLTER[2]
[1]BERGEN UNIVERSITY COLLEGE, BERGEN, NORWAY
[2]DEPT. OF INFORMATICS, UNIVERSITY OF BERGEN, BERGEN, NORWAY

1. Introduction

Designers of software specifications, experts of the application domain and programmers who are intended to implement the system need always a common language to discuss the domain. A graphical language is well suited to reason about the problem and to verify business logic with the experts of the application domain. This is because of its simplicity and universality. It's simple since a graphical representation for an ontology of a business or a hierarchy of a system is much easier to understand than their textual versions. Moreover, it's universal since people with different background can be involved in the discussion of the system architecture as long as a consistent graphical notation is used. So, a good modeling language is graph-based, formalized and sufficiently expressive to capture all the peculiarities of the universe. This is a well-known issue in Generalized Sketches (GS); dealing with formal, graphical notations. To the current state of the art in modern graphical specification languages, this goal has not been achieved; specification languages used are either semi-formal, or have a very restricted expressive power, or both [2].

There are many graphical modeling languages that are used for specification purposes in software engineering, unfortunately only few of them has proper semantics. Specifications constructed by means of informal graphical notations are often difficult to maintain due to ambiguous constructions and semantic relativism [1]; the same concepts may be understood and/or modeled differently by different designers. This leads to difficulty when the resulting software is to be expanded and integrated with other systems specified by other designers or even by the same designers.

We use GS and it's machinery to integrate and/or compare system models in a consistent way, this doesn't propose that a standard type of graphical notations should be used, but *what* should be modeled must be standardized. By superimposing signatures from different graphical notations (that have a formal semantic) and mapping them to the GS notations, we can achieve that goal.

Any specification that is sketch-able has a formal semantic and, every formal specification is sketch-able [1]. Any graphical specification, whether it is an ER diagram or a UML class diagram, can be considered as abbreviations or visualizations of sketches for a fixed signature, where the signature is its corresponding diagram type. It's important to notice that a graphical notation is a visualization on the top of a specification core. The visualization is a presentation (a user interface) of the specification, while the specification deals with the semantics of the notational constructs [3].

2. Generalized sketches

The necessity for a formal, precise formalism for software specification was the reason for a proposal made by M. Makkai [6] and later by Z. Diskin [2]. The formalism is based on already developed constructions, sketches, in Category Theory invented by Ehresmann [2]. Ehresmann's sketches are graphical presentations of categories. Makkai generalized sketches by introducing diagram predicates to them and called the result for Generalized Sketches. Diskin made GS more direct and introduced *operational* sketches specifying complex diagram operations over sketches, and thus made GS more applicable to the area of software engineering. Many of the theoretical questions and practical problems in the area are successfully approached in this framework [3].

2.1. **Definitions**[4]**.** A **graph** is composed of one or more nodes or objects that are connected by edges, it's a mathematical structure used to model pairwise relations between objects from a certain collection. A graph morphism $f : G_1 \to G_2$ maps nodes to nodes and arrows to arrows such that their incidences is preserved. A **diagram** is a visual presentation of a part of a graph. Formally, a diagram in graph G is a graph morphism $d : G_d \to G$ where G_d is a graph, the shape of d. Informally, we look at d as a sub-graph $d(G_d)$ of G. A **signature** Σ is a collection of diagram predicates which is a name together with an arity (a fixed shape.) Formally, a signature Σ is composed of a pair $< P, Arity(P) >$ where P is a set of predicate labels and $Arity(P)$ is the arity shape of P. A signature morphism is a mapping $\sigma : \Sigma_1 \to \Sigma_2$ such that $Arity(P.\sigma) = Arity(P)$, i.e. the shape graph of the predicate label is preserved. It is possible to have predicates both on nodes and on arrows - diagrams with a single arrow.

Given a signature Σ, a (Σ-)sketch $S =< G_S, D_S(P) >$ is composed of a graph G_S, the carrier graph of S, in which some diagrams D_S are marked with predicate labels $P \in \Sigma$, where $D_S(P)$ is the (possibly empty) set of all diagrams from G_S that are labeled by P. While a marked diagram is defined as the pair $< d, P >$ where $d : G_d \longrightarrow G_S$ and P is a predicate label such that $Arity(P)$ is isomorphic to d or, if $Arity(P)$ is a family, $d \in Arity(P)$. A Sketch

morphism $f : S_1 \longrightarrow S_2$ is a mapping of the carrier graphs $f : G_{S_1} \longrightarrow G_{S_2}$ such that if $(d : G_d \longrightarrow G_{S_1}) \in D_{S_1}(P)$ then $(d; f : G_d \longrightarrow G_{S_1} \longrightarrow G_{S_2}) \in D_{S_2}(P)$, i.e. the following diagram commutes.

$$\begin{array}{ccc} G_{S_1} & \xrightarrow{f} & G_{S_2} \\ \uparrow{\scriptstyle d} & \nearrow{\scriptstyle d;f} & \\ G_d & & \end{array}$$

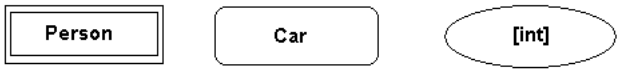
For a signature Σ, we let the space of all Σ-sketches and mappings between them be denoted by Sketch(Σ). Then a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ gives rise to a mapping between sketch spaces $\sigma_\# : Sketch(\Sigma_1) \to Sketch(\Sigma_2)$. Due to this result, we can define mappings between sketches in different signatures as a pair $\mu =< \sigma_\mu, f_\mu >: (\Sigma_1, S_1) \to (\Sigma_2, S_2)$ where $\sigma_\mu : \Sigma_1 \to \Sigma_2$ is a signature morphism and $f_\mu : S_1.\sigma_\# \longrightarrow S_2$ is a mapping of Σ_2-sketches. In this way, we can map sketches that have different graphical notations (signatures) to each other. The figure below explains this result.

$$< S_1, \Sigma_1 > \xrightarrow{\mu} < S_2, \Sigma_2 > \qquad\qquad Sketch(\Sigma_1) \xrightarrow{\sigma_\#} Sketch(\Sigma_2)$$

$$\Sigma_1 \xrightarrow{\sigma_\mu} \Sigma_2 \qquad\qquad \begin{array}{ccc} & \in & \\ S_1.\sigma_\# & \xrightarrow{f_\mu} & S_2 \end{array}$$

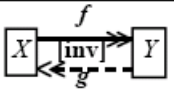
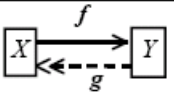
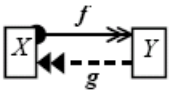
2.2. **Some examples.** The simplest diagram is a single node. A node denotes the existence of an object class in the application domain. Constraints could be applied to the node to make it compatible with the real world peculiarities. For example in Figure 1, we have three nodes which are marked with markers (predicates) from a signature, the nodes put different constraints on the object classes that are denoted by them.

FIGURE 1. Examples of nodes carrying different constraints.



An arrow is also a diagram predicate, it connects two nodes to each other and serves as a relation between the object classes. Constraints can also be applied to arrows as shown in Figure 2. Different kinds of arrow predicates denote different kinds of relationships between the object classes. Many of the arrow constraints are shown as diagram predicates that are abbreviated and drawn as markers hung on the arrows.

FIGURE 2. Example of arrows and their semantics[4]

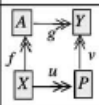
Seman-tics	*f* is partial and multi-valued, *g* is its inverse 1	*f* is single-valued 2	*f* is total (hence, *g* is covering) 3
Sketch notation			

A diagram could be more complex consisting of a few nodes and arrows from the sketch, and diagram constraints are used to denote more complex relations between them. This is needed to model complex relations between several object classes, and between different components of the system. The latter is important in the integration phase of the software life cycle, however, other graphical notations like UML and ER does not support this. The diagram that is to be marked with some label, should have the same shape as the diagram predicate from the signature. Figure 3 below shows some examples of diagram predicates.

2.3. **Advantages of GS.** Among the principal advantages of GS the following can be mentioned:

- Nice amalgamation of logical rigor and graphical evidence. GS are graph-based images yet they are precise formal specifications.
- Universality, in the precise sense of the word. It can be mathematically proved that any specification whose semantic meaning can be formalized, can also be expressed by a GS, or in other words, is sketch-able.

FIGURE 3. Diagram predicates (row 1 and 2) and a diagram operation (row 3) with their semantics in sets[1]

Name	Arity shape	Visualization (marker)	Semantics in **Sets**
inverse arrows		[inv]	for any $x \in X$, $y \in Y$ $y \in x.f \Leftrightarrow x \in y.g$
semi-commutativity		[⇑]	$u;v \leq f;g$, that is, $Dom(u) \subseteq Dom(f)$ and for any $x \in Dom(u)$, $\{p.v \mid p \in x.u\} \subseteq \{a.g \mid a \in x.f\}$
inversion of mapping		[inv]	for any $y \in Y$, $y./g := \{x \in X \mid x.f \ni y\}$, in other words, $/g = f^{-1}$

- Unifying power. Many graphical specification languages can be simulated by GS in the corresponding signature of diagram markers. That is, each graphical notation, say ERD or the different types of diagram in the UML language, corresponds to a given signature.
- Semantic capabilities. The GS language is inherently object-oriented and provides a quite natural way of specifying OO class-reference schemes.
- Easy and flexible modularization mechanism. A complex specification can be presented by a GS whose nodes are sketches and arrows are sketch mappings. This pattern can be reiterated if necessary.

3. SUMMARY

Development of a graphical specification tool with functionalities that take advantage of the GS framework is one of the major focuses of our project. Making the theory of GS more applicable to the field of software engineering constitutes another part of the project. The tool to be developed will serve as an IDE (Integrated Development Environment) with the possibility for automatic code generation and system integration for formal graphical specifications drawn based on GS formalism. We have developed a prototype [5] of a drawing tool that serves as a base for development and implementation of well-known results from GS and Category Theory, and also potential new results obtained throughout the progress of the project. Our intention is to develop techniques and methods for application of GS and its formalism to software specification. This contributes to further exploration of the practical values of the formalism in question and provides the necessary framework for building such specifications and supports the diversity of potential operations that can be applied in coordination with this universal formalism. To the current state of the art, there exist no other drawing tool for GS suitable for practical work. A program [2] has been developed for this purpose, but it is rather old and has unfortunately not been finalized, a problem that is reflected through the use of this application as it suffers from some serious bugs.

In view of the general objectives the following tasks will be addressed by our project:

- Formalization of different software description formalisms as, for example, ER-diagrams, UML-diagrams, DB-schemes, by GS.
- Investigation of the integration, the combination, and the modularization of specifications within single specifications formalisms.
- Investigation of a rigorous integration of different software description formalisms based on GS
- Design and development of tools supporting the application of GS in the field of software engineering:
  - implementing the functionality for drawing UML and ER diagrams based on GS formalism with support for mappings/translations between the two types of diagrams by mapping their signatures
  - implementation of a code-generator for generation of code in different programming languages based on the graphical specifications
  - case studies to evaluate the theory in practice.

REFERENCES

[1] Z. Diskin and Boris Kadish, Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling.
[2] Z. Diskin, Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Technical Report 9701, University of Latvia, 1997.
[3] Z. Diskin, Visualization vs. specification in diagrammatic notations: A case study with the UML.
[4] Z. Diskin, MATHEMATICS OF UML: Making the Odysseys of UML less dramatic.
[5] Ørjan Hatland, Sketcher .NET A Drawing Tool for Generalized Sketches.
[6] M. Makkai, Generalized sketches as a framework for completeness theorems.