Graph Transformation in a Nutshell

Reiko Heckel

Universität Dortmund, Germany
(on leave from Paderborn)
reiko@upb.de

Abstract. Even sophisticated techniques start out from simple ideas. Later, in reply to application needs or theoretical problems new concepts are introduced and new formalizations proposed, often to a point where the original simple core is hardly recognizable. In this paper we provide a non-technical introduction to the basic concepts of typed graph transformation systems, completed with a survey of more advanced concepts, and explain some of its history and motivations.

1 Introduction

Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular. In fact, for most activities in the software process, a variety of visual notations have been proposed, including state diagrams, Structured Analysis, control flow graphs, architectural description languages, function block diagrams, and the UML family of languages. These notations produce models that can be easily seen as graphs and thus graph transformations are involved, either explicitly or behind the scenes, when specifying how these models should be built and interpreted, and how they evolve over time and are mapped to implementations. At the same time, graphs provide a universally adopted data structure, as well as a model for the topology of object-oriented, component-based and distributed systems. Computations in such systems are therefore naturally modeled as graph transformations, too.

Graph transformation has originally evolved in reaction to shortcomings in the expressiveness of classical approaches to rewriting, like Chomsky grammars and term rewriting, to deal with non-linear structures. The first proposals, appearing in the late sixties and early seventies [11,12], are concerned with rule-based image recognition, translation of diagram languages, etc.

In this paper, we first introduce the basic concepts of graph transformation systems. Section 3 provides a high-level survey of more advanced concepts, and Section 4 gives references for further reading.

2 The Basic Approach

Modeling can be described as a two-dimensional abstraction process. The first dimension consists in building *models as representations of reality.* The second

Dagstuhl Seminar Proceedings 04101
http://drops.dagstuhl.de/opus/volltexte/2005/16

dimension is *generalization*, i.e., the extraction of concepts from concrete objets, or of rules from observed behavior. We will deal with generalization first, reserving representation issues for Sect. 2.2.

2.1 Modeling by Example

It is a matter of philosophical debate if generalizations exist in reality or if, e.g., the concept of *car* as generalization of individual cars is part of the human perception of the world. Avoiding this discission, we will illustrate two forms of generalization by means of a simple video game of *PacMan*, for didactic purposes playing the role of "reality". Later, the insights gained from this example shall be transferred to the graph-based representation to explain some of the elementary concepts of graph transformation.

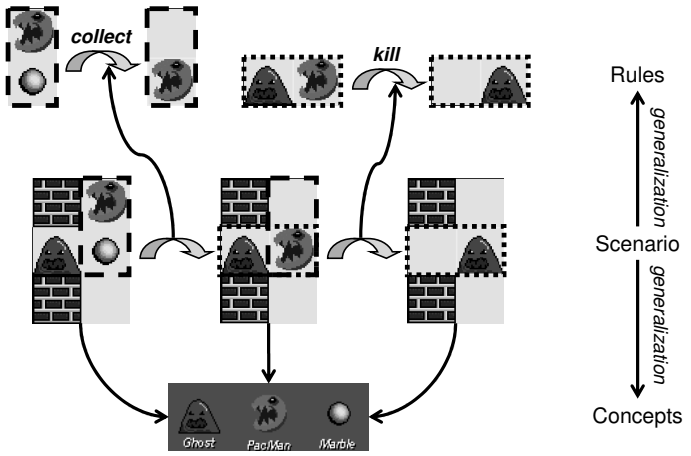

Fig. 1. From snapshots and scenarios to concepts and rules

Figure 1 exemplifies both conceptual and behavioral generalization. Our observation of the game is represented by the scenario in the middle of the figure in three successive snapshots. *Conceptual generalization* extracts three types of characters, *PacMan, Ghost*, and *Marble* shown in the bottom, each of which has several instances in the scenario. This conceptual generalization and the corresponding relation between a concept (type) and its instances is the first basic idea.

From the observed transformations, *behavioral generalization* extracts the rules in the top, encapsulating the changes from one snapshot to the next. Rules can be derived systematically by determining their scope in the transformation and cutting off (abstracting from) the irrelevant context. For the rules *collect* and *kill* in the top of the figure, their scope is given by the areas in the dashed and dotted boxes, respectively.

The actual game, from which the snapshots are taken, has been produced using the *StageCast* visual programming environment for video games. The environment evolved out of a didactic experiment on teaching the basics of programming to children from the age of 7. The idea of extracting rules as general behavior descriptions from sample state transformations is called *programming by example* and represents the main didactic tool of the *StageCast* environment. It also provides a perfect example of the second basic idea of graph transformation: the definition of rules as specifications of state transformations.

In the following, we shall turn back to the first dimension of modeling, transferring the generalizations described above from "reality" to its *representation* in a model. We will use graphs as means to represent snapshots, concepts, and rules—the third basic idea of the approach.

2.2 Type and Instance Graphs

Graphs provide the most basic mathematical model for entities and relations. A graph consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. Like the graph G in the upper right of Fig. 2, graphs can represent snapshots by modeling concrete entities as vertices and relations between theses entities as edges. In our model, vertices *P:PacMan, G:Ghost M:Marble* represent the corresponding characters in the snapshot. Another type of vertex is used to represent fields, i.e., the open spaces in the snapshot where characters can be located. Edges represent the current location of characters as well as the neighborhood relation of fields.

In modeling the snapshot we have implicitly assumed that vertices have a type, like $F1$ to $F4$ having type $Field$. The type of a vertex (or an edge) represents the conceptual generalization of the corresponding real-world entity. Like an individual snapshot, also a collection of interrelated concepts may be represented as a graph. Figure 2 in the bottom right shows an example of a type graph representing the conceptual structure of the PacMan game and providing the types for the instance graph in the top.

The relation between concepts and their occurrences in snapshots is formally captured by the notion of *typed graphs*: A fixed *type graph* TG represents the type (concept) level and its *instance graphs* the individual snapshots. This distinction is a recurring pattern, like in class and objects, data base schema and states, XML schema and documents, etc.

We use the notation of class and object diagrams in the Unified Modeling Language (UML) to visualize type and instance graphs and their relationship, i.e., $o : C$ represents a vertex o (an object) of type C (a class). In addition to vertices and edges, graphs may contain attributes to store values of pre-defined data types. In our example, this notion is used to represent the number of marbles PacMan has collected before the current snapshot. Also attributes have a type-level declaration $a : T$, where a is the name of the attribute and T is the data type, and an instance-level occurrence $a = v$ where attribute a is assigned value v.

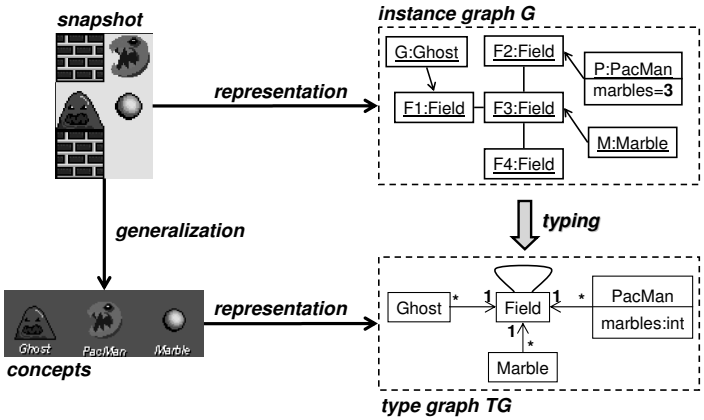

Fig. 2. Type and instance graph of the PacMan example

The relation between type and instance level is subject to the usual compatibility conditions, i.e.,

- for each vertex $o : C$ in the instance graph there must a vertex type C in the type graph;
- for each edge between objects $o_1 : C_1$ and $o_2 : C_2$ there must be a corresponding edge type in the type graph between vertex types C_1 and C_2;
- for each attribute value $a = v$ associated with a vertex $o : C$ in an instance graph, there must be a corresponding declaration $a : T$ in vertex type C such that v is of data type T;

2.3 Rules and Transformations

Having represented snapshots as instance graphs over a type graph built as a representation of the game's concepts, we are turning to the genuine core of the approach: the specification of instance graph transformations by means of rules. Following the idea of extracting rules from transformation scenarios, Fig. 3 shows a graph representation of the behavioral generalization illustrated in the upper part of Fig. 1.

The generalization is achieved by focussing on the relevant subgraph in the source state and observing its changes in the target state. But besides cutting off context, we also abstract from the concrete attribute values replacing, e.g., 3 in G_0 by m and 4 in G_1 by $m + 1$. The resulting rules are shown in the top of Fig. 4 and 5, with the rules for moving *PacMan* and *Ghost* in the bottom.

More formally, fixing a type graph TG, a *graph transformation rule* $p : L \to R$ consists of a name p and a pair of instance graphs over TG whose structure is compatible, i.e., vertices with the same identity in L and R have the same type and attributes, and edges with the same identity have the same type, source,

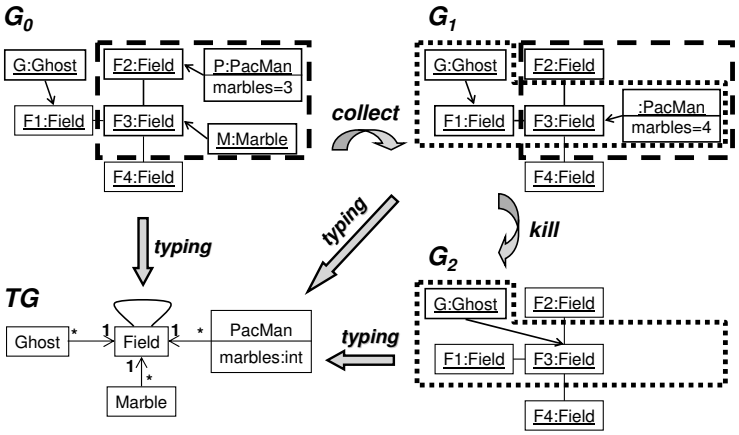

Fig. 3. From scenarios to rules: graph representation

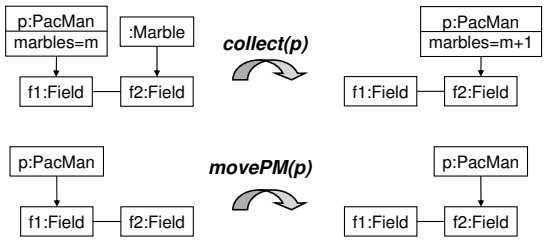

Fig. 4. Rules for PacMan

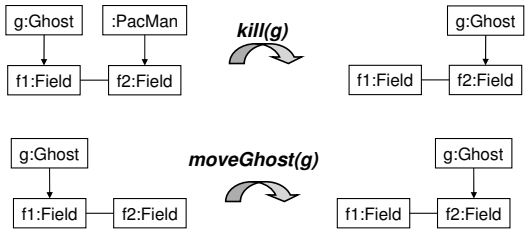

Fig. 5. Rules for Ghosts

and target. The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

But rules do also have a constructive meaning, besides being generalizations of transformations. They *generate* transformations by replacing in a given graph an occurrence of the left-hand side with a copy of the right-hand side. Thus, a *graph transformation* from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is performed in three steps.

1. *Find* an occurrence o_L of the left-hand side L in the given graph G.
2. *Delete* from G all vertices and edges matched by $L \setminus R$.
3. *Paste* to the result a *copy* of $R \setminus L$, yielding the derived graph H.

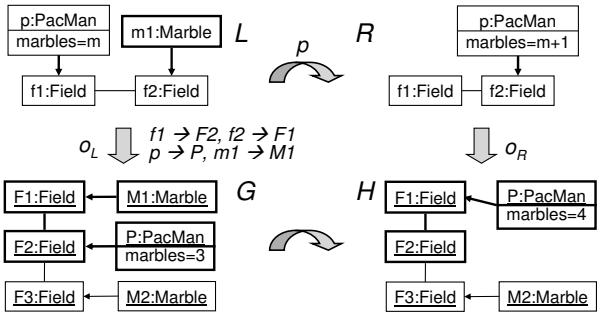

Fig. 6. Transformation step using rule *collect*

In Fig. 6 the occurrence o_L of the rule's left-hand side is indicated next to the left downward arrow. The variable m representing the value of the marble attribute before the step, is assigned value 3. The transformation deletes the edge from *PacMan* P to Field $F2$, because it is matched by the edge from $f1$ to p in L, which does not occur in R. The same applies to the value 3 of the *marbles* attribute of vertex P. To the graph obtained after deletion, we paste a copy of the edge from p to $f2$ in R. The occurrence o_L tells us where this edge must be added, i.e., to the images of p and $f2$, P and $F1$, respectively. At the same time, the new attribute value $marbles = 3 + 1 = 4$ is computed from the memorized old value $m = 3$.

However, this is not the only possibility for applying this rule. Another option would be to map $f1 \mapsto F2, f2 \mapsto F3, p \mapsto P, m \mapsto M2$, collecting the lower marble instead. Also, we could have chosen to apply the *movePM* rule. That means, there are two causes of non-determinism: choosing the rule and the occurrence at which it is applied.

The total behavior of our *PacMan* game is given by the set of all sequences of consecutive transformation steps $G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n$ using the rules of the game and starting from a valid instance graph G_0. As a simple example, we

recall the two-step sequence in Fig. 3 which is re-generated by application of the two previously extracted rules. Note that all graphs of a sequence must be valid instances of the fixed type graph TG.

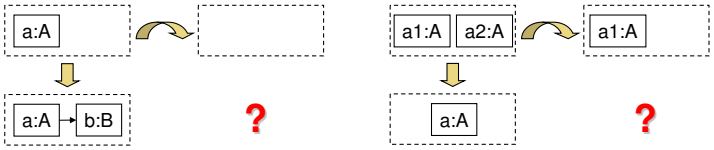

Fig. 7. More difficult examples

The example of Fig. 6 is not entirely representative of the problems that may be caused by deleting elements in a graph during step 2. In fact, we have to make sure that the remaining structure $G \setminus o(L \setminus R)$ is still a graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. The problem is exemplified in its simplest form by the step in Fig. 7 on the left. A related problem is depicted on the right, where we observe a conflict between deleting vertex $a : A$ as required by $a1 : A$ and preserving it as suggested by $a2 : A$.

In both cases, there exist a radical and a conservative solution: The first gives priority to deletion, deleting the vertex along with the dangling edge in the left example and vertex a in the right example of Fig. 7. Both may lead to surprising and undesired effects and may require additional control to restrict rule applications to the intended cases.

The safer alternative consists in formulating standard applications conditions which exclude the depicted situations as valid transformations. This is achieved by the *gluing conditions* of the algebraic (or double-pushout) approach to graph transformation, which provides the basis for the presentation in this text.

Next we discuss some extensions to this basic approach.

3 Advanced concepts

The phenomenon of "dangling edges" is caused by the fact that a node in a graph may, in general, have an unknown number of connections. This is in contrast with, e.g., the rewriting of strings where the linear structure provides exact information about the connections of any substring. The additional complication has led to a number of extensions of the basic approach, some of which shall be discussed below.

3.1 Constraints

One way of solving the problem is to constrain the number of connections. However, type graphs are not expressive enough to define such restrictions. For example, in order to model the PacMan gamebord it makes sense to require that

each *Ghost*, *Pacman*, or *Marble* vertex is linked to exactly one *Field* vertex. Such constraints need to be expressed by additional cardinality annotations as shown in the type graph of Fig. 2.

More complex constraints could deal with the (non-) existence of certain patterns, including paths, cycles, etc. They can be expressed in terms of logic formulae or as graphical constraints. An example for the latter is given in Fig. 8. The constraint expresses by means of a *forbidden subgraph* that there must not be a Ghost and a PacMan situated at the same Field. In order to satisfy the constraint, a graph G must not contain a subgraph isomorphic to it. In first order logic, the same property could read $\neg\exists g : Ghost; p : PacMan; f : Field.\ at(g, f) \wedge at(p, f)$.

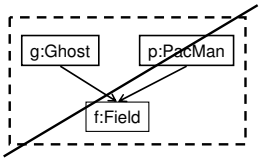

Fig. 8. A graphical constraint

Constraints restrict the set of admissible instance graphs and can be used to control the transformation process by ruling out transformations leading to non-admissible graphs. This is comparable to the integrity mechanism in a data base management system which checks the validity of constraints after each update, but before the new state is committed.

3.2 Multi-objects

Also more complex operations, like "delete all Ghosts located on Fields directly reachable from PacMan's current position", can only be specified in the presented approach if the number of reachable fields is known in advance. This is not the case in our example. Thus, for expressing such universally quantified operations, we have to adopt the additional concept of *multi object* from UML object diagrams.

A multi object like $f2$ in the rule of Fig. 9 stands for the maximal set of objects that have the specified connections to the fixed objects in the rule, in our case all the neighboring fields of the match $F2$ of $f1$. Similarly, $g : Ghost$ stands for all ghosts in G located at these fields. The example shows that the universal quantification extends smoothly to the actions of the rule, i.e., the deletion of the ghosts.

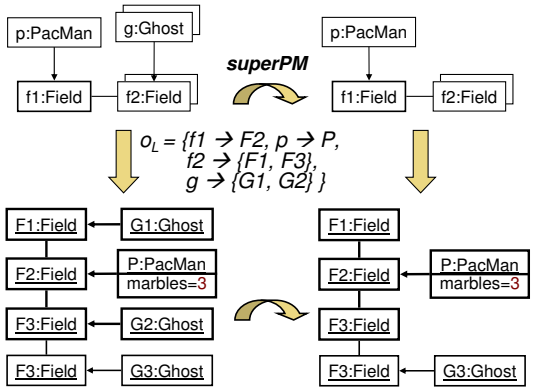

Fig. 9. Rule application with multi object

3.3 Application conditions

Generalizing the pre-defined gluing conditions of the algebraic approach, user defined application conditions are used, for example, to "sense" the existence or non-existence of connections in the vicinity of the occurrence of the rule's left hand side. Figure 10 shows an improved rule *movePM* checking that there is no marble in the place PacMan is moving to. Graphically, this is indicated by the crossed out vertex in the left-hand side.

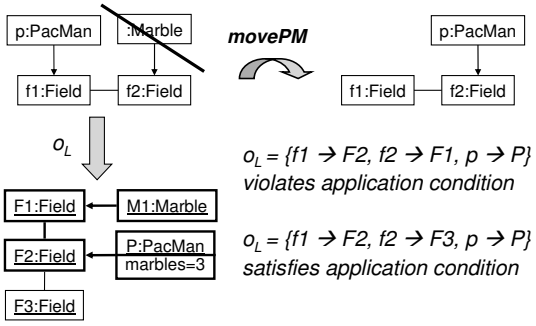

Fig. 10. Rule application with negative application condition

Intuitively, the rule is applicable at an occurrence if this cannot be extended to the forbidden elements. That means, the applicability does not only depend on the graph G, but also on the chosen occurrence, as indicated in the figure.

3.4 Control conditions

Beside extensions that restrict the applicability of individual rules, global control structures lead to *programmed graph transformations*. Here, rules embedded them into imperative programming constructs or a visual control flow language, as shown in Fig. 11 where UML activity diagrams are used to express, respectively, the order of rule applications to individual PacMan and Ghost vertices.

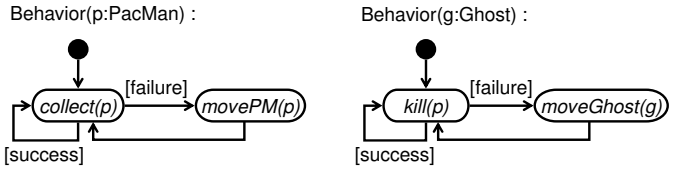

Fig. 11. Controlling rules by activity diagrams

Special conditions *[failure]* and *[success]* are introduced to make the control flow dependent of the (non-)applicability of rules. Another notable feature is the possibility of passing parameters to rules. In our example it is intended that, e.g., the order of collect and movePM operations is understood locally for any individual PacMan vertex.

4 Further Reading

Fundamental approaches to graph transformation [13] include the algebraic or double-pushout (DPO) approach [7], the node-label controlled (NLC) [10] approach, and the PROGRES approach [14] which represents the first major application of graph transformation to software engineering [8].

The simple core model introduced in Section 2 is based on (a set-theoretic presentation of) the double-pushout approach [7]) whose features are common to most graph transformation approaches. Generally, we distinguish two fundamentally different views of graph transformation referred to as the *gluing* and the *connecting* approach. They differ for the mechanism used to embed the right-hand side of the rule in the context (the structure left over from the given graph after deletion): In a gluing approach like DPO, the new graph is formed by gluing the right-hand side with the context along common vertices. In a connecting approach like NLC, the embedding is realized by a disjoint union, with as many new edges as needed to connect the right-hand side with the rest of the graph.

This feature, which provides one possible answer to a fundamental problem, the replacement of substructures in an unknown context, is known in software engineering-oriented approaches by the name of *set nodes* or *multi objects*, e.g., PROGRES [14,15], a language and tool for *PROgrammed Graph REwriting Systems*, and FUJABA [1], an environment for round trip engineering between UML

collaboration diagrams and Java based on graph transformation as the operational model.

Both approaches have extended the basic approach by *programmed* transformations, concerned with controlling the (otherwise non-deterministic) rewrite process, as well as *application conditions*, restricting the applicability of individual rules, as well as structural constraints over graphs, comparable to invariants or integrity constraints in data bases, deal with the (non-) existence of certain patterns, including paths, cycles, etc. They are expressed as cardinalities, in terms of first- or higher-order logic, or as graphical constraints. The latter are also supported by AGG [9], which implements the algebraic approach by means of a rule interpreter and associated analysis techniques.

For recent surveys on graph transformations and its applications see the handbooks [13,4,6] the survey paper [2] and the tutorial paper [3].

References

1. From UML to Java and Back Again: The Fujaba homepage. `www.upb.de/cs/isileit`.
2. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999. `http://www.uni-paderborn.de/cs/ag-engels/Papers/AndriesSCP99.pdf`.
3. L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proc. of the First International Conference on Graph Transformation (ICGT 2002), Barcellona, Spain*, pages 402–429, oct 2002.
4. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.
5. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
6. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. World Scientific, 1999.
7. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
8. G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software specification using graph grammars. *Computing*, 31:317–346, 1983.
9. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In Engels et al. [5], pages 551 – 601.
10. D. Janssens and G. Rozenberg. On the structure of node-label controlled graph grammars. *Information Science*, 20:191–216, 1980.
11. J. L. Pfaltz and A. Rosenfeld. Web grammars. *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
12. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
13. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

14. A. Schürr. Programmed graph replacement systems. In Rozenberg [13], pages 479 – 546.
15. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [5], pages 487–550.