Chapter 1

GRAPH TRANSFORMATION UNITS AND MODULES

The aim of this chapter is to introduce the notions of transformation units and transformation modules as means of constructing large graph transformation systems from small ones in a structured and systematic way. A transformation unit comprises a set of rules, descriptions of initial and terminal graphs, and a control condition. Moreover, it may use other transformation units for structuring purposes. Its semantics is a binary relation between initial and terminal graphs which is given by interleaving ordinary direct derivation steps with calls of imported transformation units. Putting transformation units together yields transformation modules. A module may have an import interface consisting of formal parameter units and an export interface consisiting of the units that are made available to the environment. The formal parameter units can be instantiated by exported units of other modules. The introduced framework is independent of a particular graph transformation approach and, therefore, it may enhance the usefulness of graph transformations in many contexts.

1.1 Introduction

The significance of graphs and rules in many areas of computer science is evident: On the one hand, graphs constitute appropriate means for the description of complex relationships between objects. Trees, Petri nets, circuit diagrams, data flow graphs, state charts, and entity-relationship diagrams are some typical examples. On the other hand, rules are used to describe "permitted" actions on objects as, for example, in the areas of functional and logic programming, formal languages, algebraic specification, theorem proving, and rule-based systems.

The intention of bringing graphs and rules together—motivated by several application areas—has led to the theory of graph grammars and graph transformation (see volume I of the Handbook and [1–7] for a survey). A wide spectrum of approaches exists within this theory and some of them are implemented (see, for example, PROGRES [8,9], Graph$^{\mathrm{Ed}}$ [10], Dactl [11], and AGG [12,13]).

With the aim of enhancing the usefulness of graph transformation, we introduce approach-independent structuring methods for building up large systems of graph transformation rules from small pieces. The methods are based on the notion of a transformation unit and its interleaving semantics. A transformation unit is allowed to use other units such that a system of graph transformation rules can be structured and existing transformation units can be re-used.

To make the structuring more flexible, a cluster of transformation units can be encapsulated into a module. This allows to distinguish between main units to be exported and made available to the environment and auxiliary units that are hidden in the body. Moreover, a module can import formal parameter units that allow to leave parts of a system unspecified for later actualization. Transformation units and modules are basic concepts of the new graph and rule centered language GRACE that is being developed by researchers from Berlin, Bremen, Erlangen, München, Oldenburg, and Paderborn (see also [6, 14–16]). Nevertheless, the notion is meaningful in its own right because – independently of GRACE – it can be employed as a structuring principle in most graph transformation approaches one encounters in the literature where graph transformation is often called graph rewriting.
The chapter is organized as follows. In section 1.2 we discuss the notion of a transformation unit together with its interleaving semantics. In section 1.3, the concepts of a transformation unit are illustrated with the sample specification of a shortest-path algorithm. Finally, we introduce the notion of transformation modules and their composition in section 1.4. The chapter ends with some concluding remarks.

1.2 Transformation Units

The key operation in graph transformation approaches is the direct derivation being the transformation of a graph into a graph by applying a rule. In other words, each rule yields a binary relation on graphs. Hence, each set of rules specifies a binary relation on graphs by iterated rule applications. This derivation process is highly non-deterministic in general and runs on arbitrary graphs which is both not always desirable. For example, if one wants to generate graph languages, one may start in a particular axiom and end with certain terminal objects only. Or if a more functional behaviour is required, one may prefer to control the derivation process and to cut down its non-determinism. The latter can be achieved by control mechanisms for the derivation process like application conditions or programmed graph transformation (see, e.g., [17–27], cf. also [28] for regulation concepts in string grammars) and the former by the use of graph class expressions that specify subclasses of graphs. Moreover, in practical cases, one may have to handle hundreds or thousands of rules which cannot be done in a transparent and reasonable way without a structuring principle.
To cover all these aspects, we introduce the notion of a transformation unit that allows to specify new rules, initial and terminal graphs, as well as a control condition, and to import other transformation units. Semantically, a transfor-

mation unit describes a graph transformation, i.e. a binary relation on graphs given by the interleaving of the imported graph transformations with each other and with rule applications. Moreover, interleaving sequences must start in initial graphs, end in terminal graphs and satisfy the control condition. If nothing is imported, the interleaving semantics coincides with the derivation relation.
To make the concept independent of a particular graph transformation framework, we assume an abstract notion of a graph transformation approach comprising a class of graphs, a class of rules, a rule application operator, a class of graph class expressions, and a class of control conditions. The semantic effect of control conditions depends on so-called environments. In this way, it can be defined without forward reference to transformation units.
Examples of graph class expressions and control conditions are given after the introduction of transformation units and their interleaving semantics. At the end of this section, we consider a certain class of control conditions which consists of languages over rules and transformation units and point out its relation to interleaving sequences.

1.2.1 Graph Transformation Approach

A *graph transformation approach* is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$ where

- $\mathcal{G}$ is a class of *graphs*,
- $\mathcal{R}$ is a class of *rules*,
- $\Rightarrow$ is a *rule application operator* yielding a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ for every $r \in \mathcal{R}$,
- $\mathcal{E}$ is a class of *graph class expressions* such that each $e \in \mathcal{E}$ specifies a subclass $SEM(e) \subseteq \mathcal{G}$, and
- $\mathcal{C}$ is a class of *elementary control conditions* over some set ID of identifiers such that each $c \in \mathcal{C}$ specifies a binary relation $SEM_E(c) \subseteq \mathcal{G} \times \mathcal{G}$ for each mapping $E\colon ID \longrightarrow 2^{\mathcal{G}\times\mathcal{G}}$.[a]

A pair $(G, G') \in \Rightarrow_r$, usually written as $G \Rightarrow_r G'$, establishes a *direct derivation* from G to G' through r. For a set $P \subseteq \mathcal{R}$ the union of all relations $\Rightarrow_r$ $(r \in P)$ is denoted by $\Rightarrow_P$ and its reflexive and transitive closure by $\Rightarrow_P^*$. A pair $(G, G') \in \Rightarrow_P^*$, usually written as $G \Rightarrow_P^* G'$, is called a *derivation* from G to G' over P. A mapping $E\colon ID \longrightarrow 2^{\mathcal{G}\times\mathcal{G}}$ is called an *environment*. In the

[a]The power set of a set S is denoted by 2^S.

following, we will use boolean expressions over $\mathcal{C}$ as *control conditions* with elementary control conditions as basic elements and disjunction, conjunction, and negation as boolean operators. Moreover, we make use of the constant *true*. The semantic relations of elementary control conditions are easily extended to boolean expressions by

$$\begin{aligned}
&SEM_E(true) = \mathcal{G} \times \mathcal{G},\\
&SEM_E(e_1 \vee e_2) = SEM_E(e_1) \cup SEM_E(e_2),\\
&SEM_E(e_1 \wedge e_2) = SEM_E(e_1) \cap SEM_E(e_2),\\
&SEM_E(\overline{e}) = \mathcal{G} \times \mathcal{G} - SEM_E(e).
\end{aligned}$$

The set of control conditions over $\mathcal{C}$ is denoted by $\mathcal{B}(\mathcal{C})$.
Note that we refer to the meaning of graph class expressions and control conditions by the overloaded operator *SEM*. This should do no harm because it is always clear from the context which is which.
All the graph grammar and graph transformation approaches one encounters in the literature provide notions of graphs and rules and a way of directly deriving a graph from a graph by applying a rule (cf. e.g. [8, 10, 18, 20, 29–33]). Therefore, all of them can be considered as graph transformation approaches in the above sense if one chooses the components $\mathcal{E}$ and $\mathcal{C}$ in some standard way. The singleton set $\{all\}$ with $SEM(all) = \mathcal{G}$ may provide the only graph class expression, and the class of elementary control conditions may be empty. Non-trivial choices for $\mathcal{E}$ and $\mathcal{C}$ are discussed in subsections 1.2.4 and 1.2.5.

1.2.2 Transformation Units

A transformation unit encapsulates a specification of initial graphs, a set of identifiers referring to transformation units to be used, a set of rules, a control condition, and a specification of terminal graphs.
Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$ be a graph transformation approach. A *transformation unit* over $\mathcal{A}$ is a system $trut = (I, U, R, C, T)$ where $I, T \in \mathcal{E}$, U is a finite set of identifiers, $R \subseteq \mathcal{R}$ is a finite set of rules, and $C \in \mathcal{B}(\mathcal{C})$. The components of *trut* may be denoted by U_{trut}, I_{trut}, R_{trut}, C_{trut}, and T_{trut}, respectively. The class of all transformation units over $\mathcal{A}$ is denoted by $\mathcal{T}_{\mathcal{A}}$. The component U may be seen as a set of formal parameters that can be instantiated by transformation units.
To keep the technicalities simple, one may assume that only defined transformation units are imported. Hence, initially, U must be chosen as the empty set yielding unstructured transformation units without import. Such transformation units of level 0 may be used in transformation units of level 1. Iteratively, one obtains a transformation unit of level $i + 1$ for some $i \in \mathbb{N}$ if one imports transformation units up to level i. In this way, the import structures of

transformation units become acyclic. This provides a principle of hierarchical structuring. The case of an arbitrary import structure is studied in [16].
If I specifies a single graph (cf. item 1 of 1.2.4), U is empty, and C is the constant *true*, one gets the usual notion of a graph grammar (in which approach ever) as a special case of transformation units.

1.2.3 Interleaving Semantics

The semantics of a transformation unit is a graph transformation, i.e. a binary relation on graphs containing a pair (G, G') of graphs if, first, G is an initial graph and G' is a terminal graph, second, G' can be obtained from G by interleaving direct derivations with the graph transformations specified by the used transformation units, and third, the pair is allowed by the control condition.
Let $\mathit{trut} = (I, U, R, C, T)$ be a transformation unit over the graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$. Assume that the set ID of identifiers associated to $\mathcal{C}$ contains the disjoint union of U and R. Let the interleaving semantics $\mathit{SEM}(t) \subseteq \mathcal{G} \times \mathcal{G}$ for $t \in U$ be already defined. Let $E(\mathit{trut})\colon \mathit{ID} \to 2^{\mathcal{G} \times \mathcal{G}}$ be defined by $E(\mathit{trut})(r) = \Rightarrow_r$ for $r \in R$, $E(\mathit{trut})(t) = \mathit{SEM}(t)$ for $t \in U$, and $E(\mathit{trut})(id) = \{\}$, otherwise. Then the *interleaving semantics* $\mathit{SEM}(\mathit{trut})$ of *trut* consists of all pairs $(G, G') \in \mathcal{G} \times \mathcal{G}$ such that

1. $G \in \mathit{SEM}(I)$ and $G' \in \mathit{SEM}(T)$,
2. there are graphs $G_0, \ldots, G_n \in \mathcal{G}$ with $G_0 = G$, $G_n = G'$, and for $i = 1, \ldots, n$, $G_{i-1} \Rightarrow_r G_i$ for some $r \in R$ or $(G_{i-1}, G_i) \in \mathit{SEM}(t)$ for some $t \in U$,
3. $(G, G') \in \mathit{SEM}_{E(\mathit{trut})}(C)$.

The sequence of graphs in point 2 is called an *interleaving sequence in trut from G to G'*. Let $\mathit{RIS}_{\mathit{trut}}$ denote the binary relation given by interleaving sequences, i.e. $\mathit{RIS}_{\mathit{trut}} = (\Rightarrow_R \cup \bigcup_{t \in U} \mathit{SEM}(t))^*$. Then the interleaving semantics of *trut* is defined as the intersection of $\mathit{RIS}_{\mathit{trut}}$ with $\mathit{SEM}(I) \times \mathit{SEM}(T)$ and $\mathit{SEM}_{E(\mathit{trut})}(C)$. Note that all three relations may be incomparable with each other. For example, $(G, G') \in \mathit{SEM}_{E(\mathit{trut})}(C)$ does not imply in general that there is an interleaving sequence in *trut* from G to G', and vice versa.
A control condition C specifies a binary predicate depending on other binary graph relations through the notion of environments, but independent of a particular transformation unit. As a component of *trut*, only the *environment of trut* given by $E(\mathit{trut})$ is effective, meaning that C can restrict the semantics by specifying certain properties of the direct derivation relations of rules in

trut, the interleaving semantics of imported transformation units, and the interrelation of all of them. If transformation units are employed as structuring concepts in a specification language, it would be reasonable to assume that rules may be named and that only their names belong to the set of identifiers rather than the rules themselves. But the naming of rules is not needed here.
The definition of the interleaving semantics follows the recursive definition of transformation units. Hence, its well-definedness follows easily by an induction on the import structure, i.e. on the levels of transformation units.
Initially, if U is empty, an interleaving sequence is just a derivation such that one gets in this case

$$SEM(trut) = \Rightarrow_R^* \cap (SEM(I) \times SEM(T)) \cap SEM_{E(trut)}(C).$$

In other words, interleaving semantics generalizes the ordinary semantics of sets of rules given by derivations.
If I is a single graph (specifying itself as initial graph in the sense of 1.2.4.1), the first component of the interleaving semantics of *trut* is insignificant. Then all second components form a graph language that can be considered as the language generated by the transformation unit, i.e.

$$L(trut) = \{G \in SEM(T) \mid (I, G) \in SEM(trut)\}.$$

In this case, the transformation unit is called *language-generating.* If, furthermore, U is empty and C is *true*, *trut* is a graph grammar (cf. 1.2.2), and its generated language consists, as usual, of all terminal graphs derivable from the initial graph, i.e.

$$L(trut) = \{G \in SEM(T) \mid I \Rightarrow_R^* G\}.$$

In this sense, the interleaving semantics covers the usual notion of graph languages generated by graph grammars.
The interleaving semantics of a transformation unit is defined for any choice of the imported transformation units. If one fixes the import, the interleaving semantics is a binary relation on graphs. But if one does not fix the import, the interleaving semantics can be considered as an operator yielding a binary relation on graphs for each choice of binary relations on graphs for the import parameters.

1.2.4 Graph Class Expressions

There are various standard ways to choose graph class expressions that can be combined with many classes of graphs and hence used in many graph transformation approaches.

1. In most cases, one deals with some kind of finite graphs with some explicit representations. Then single graphs (or finite enumerations of graphs) may serve as graph class expressions. Semantically, each graph G represents itself, i.e. $\mathit{SEM}(G) = \{G\}$. The axiom of a graph grammar is a typical example of this type.

2. A graph G is *reduced* with respect to a set of rules $P \subseteq \mathcal{R}$ if there is no $G' \in \mathcal{G}$ with $G \Rightarrow_r G'$ and $r \in P$. In this way, P can be considered as a graph class expression with $\mathit{SEM}(P) = \mathit{RED}(P)$ being the set of all reduced graphs with respect to P. Reducedness is often used in term rewriting and term graph rewriting as a halting condition.

3. If $\mathcal{G}$ is a class of labelled graphs with label alphabet Σ, then a set $T \subseteq \Sigma$ is a suitable graph class expression specifying the graph class $\mathit{SEM}(T) = \mathcal{G}_T$ consisting of all graphs labelled in T only. This way of distinguishing terminal objects is quite popular in formal language theory.

4. Graph theoretic properties can be used as graph class expressions. In particular, monadic second order formulas for directed graphs, hypergraphs or undirected graphs are suitable candidates (see e.g. [31]).

5. Graph schemata, as used in the graph transformation approach PROGRES are graph class expressions that allow to specify generic graph classes (see [9] for more details).

6. A language-generating transformation unit *trut* as introduced in 1.2.3 can be used as a graph class expression with $\mathit{SEM}(\mathit{trut}) = L(\mathit{trut})$.

1.2.5 Control Conditions

A control condition is meant to restrict the derivation process. A typical example is to allow only interleaving sequences where the sequences of applied rules and called transformation units belong to a particular control language. Therefore, a regular expression over the set of identifiers can be considered as a control condition because it specifies a language. In general, every description of a binary relation on graphs may be used as a control condition. Here, we give some examples.

1. Let $E\colon \mathit{ID} \to 2^{\mathcal{G}\times\mathcal{G}}$ be an environment. Then E can be extended to the set of languages over ID, i.e. the power set of the set of strings over ID in a natural, straight-forward way. $\widehat{E}\colon 2^{\mathit{ID}^*} \to 2^{\mathcal{G}\times\mathcal{G}}$ is defined by $\widehat{E}(L) = \bigcup_{w\in L} \overline{E}(w)$ for $L \subseteq \mathit{ID}^*$ where $\overline{E}\colon \mathit{ID}^* \to 2^{\mathcal{G}\times\mathcal{G}}$ is recursively

given by $\overline{E}(\lambda) = \Delta\mathcal{G}$, and $\overline{E}(xv) = E(x) \circ \overline{E}(v)$ for $x \in \mathit{ID}$ and $v \in \mathit{ID}^*$.[b] Hence, L can be used as control condition with $\mathit{SEM}_E(L) = \widehat{E}(L)$ for all $E\colon \mathit{ID} \to 2^{\mathcal{G}\times\mathcal{G}}$. In this case, the class of elementary control conditions is 2^{ID^*}. We refer to conditions in this class as *control conditions of language type.*

2. As a consequence of point 1, every grammar, automaton or expression x which specifies a language $L(x)$ over ID can serve as a control condition with $\mathit{SEM}_E(x) = \mathit{SEM}_E(L(x)) = \widehat{E}(L(x))$ for all environments E.

3. In particular, the class of regular expressions over ID can be used for this purpose. For explicit use below, $\mathit{REG}(\mathit{ID})$ is recursively given by $\emptyset, \epsilon \in \mathit{REG}(\mathit{ID})$, $\mathit{ID} \subseteq \mathit{REG}(\mathit{ID})$, and $(e_1\,;\,e_2)$, $(e_1 \mid e_2)$, $(e^*) \in \mathit{REG}(\mathit{ID})$ if $e, e_1, e_2 \in \mathit{REG}(\mathit{ID})$. In order to omit parentheses, we assume that * has a stronger binding than ; which in turn has a stronger binding than $\mid$.[c] The language $L(e)$ specified by some regular expression e is defined as $L(\emptyset) = \{\}$, $L(\epsilon) = \{\lambda\}$, $L(\mathit{id}) = \{\mathit{id}\}$ for all $\mathit{id} \in \mathit{ID}$, $L(e_1\,;\,e_2) = L(e_1)\cdot L(e_2)$, $L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$ and $L(e^*) = L(e)^*$.[d]

4. A pair (G, G') of graphs is *reduced* with respect to a control condition $c \in \mathcal{C}$ and an environment E if there is no graph G'' with $(G', G'') \in \mathit{SEM}_E(c)$. In this way, $c\,!$ defines a control condition where $\mathit{SEM}_E(c\,!)$ is the subset of $\mathit{SEM}_E(c)$ consisting of all reduced pairs with respect to c and E.

5. A special case of such a control condition corresponds to the notion of reduced graphs and is given by a set of rules R. $R\,!$ means that the rules of R must be applied as long as possible.

6. Each transformation unit *trut* can serve as a control condition because semantically it specifies a binary relation on graphs. For each environment E, the semantics of the control condition *trut* is given by the semantics of *trut*, i.e. by all pairs (G, G') of graphs such that G can be transformed into G' with the transformation unit *trut*.

[b] $\Delta\mathcal{G}$ denotes the identity relation on $\mathcal{G}$. Given $\rho, \rho' \subseteq \mathcal{G}\times\mathcal{G}$, the sequential composition of ρ and ρ' is defined as usual by $\rho\circ\rho' = \{(G, G'') \mid (G, G') \in \rho \text{ and } (G', G'') \in \rho' \text{ for some } G' \in \mathcal{G}\}$.

[c] While $\emptyset$ denotes the empty set $\{\}$, the expression ϵ denotes the regular set $\{\lambda\}$. We prefer a direct reference to $\{\lambda\}$ rather than to use $\emptyset^*$.

[d] Given $L, L' \subseteq \mathit{ID}^*$, the concatenation of L and L' is defined as usual by $L \cdot L' = \{ww' \mid w \in L\ ,\ w' \in L'\}$, and the Kleene closure of L is defined as usual by $L^* = \bigcup_{i=0}^{\infty} L^i$ where $L^0 = \{\lambda\}$ and $L^{i+1} = L \cdot L^i$.

7. Each pair $(e_1, e_2) \in \mathcal{E} \times \mathcal{E}$ defines a binary relation on graphs by $SEM((e_1, e_2)) = SEM(e_1) \times SEM(e_2)$ and, therefore, it can be used as a control condition which is independent of the choice of an environment, i.e. $SEM_E((e_1, e_2)) = SEM((e_1, e_2))$ for all environments E. In particular, let $trut = (I, U, R, C, T)$ be a transformation unit. Then the pair (I, T) forms a control condition.

8. For readers familiar with the graph transformation language PROGRES, it shall be mentioned that the deterministic and non-deterministic control structures of PROGRES serve as control conditions. They allow to define imperative commands over control conditions.

9. Another type of control conditions are priorities among the rules of a transformation unit. See Litovski and Métivier [24] for a particular approach of this kind.

As the following observation shows, the semantic relations given by regular expressions as control conditions can be constructed easily according to the recursive structure of regular expressions without reference to the languages generated by the expressions.

Observation 1.2.1
For all environments E, all $id \in ID$ and all $e, e_1, e_2 \in REG(ID)$ the following holds.

1. $SEM_E(\emptyset) = \{\}$.
2. $SEM_E(\epsilon) = \Delta\mathcal{G}$.
3. $SEM_E(id) = E(id)$.
4. $SEM_E(e_1\,;\,e_2) = SEM_E(e_1) \circ SEM_E(e_2)$.
5. $SEM_E(e_1 \,|\, e_2) = SEM_E(e_1) \cup SEM_E(e_2)$.
6. $SEM_E(e^*) = SEM_E(e)^*$.[e]

Proof

1. $SEM_E(\emptyset) =_{def} \widehat{E}(\{\}) =_{def} \{\}$.[f]

[e] For $\rho \subseteq \mathcal{G} \times \mathcal{G}$, ρ^* denotes the reflexive and transitive closure of ρ that is $\rho^* = \bigcup_{i=0}^{\infty} \rho^i$ where $\rho^0 = \Delta\mathcal{G}$ and $\rho^{i+1} = \rho \circ \rho^i$.

[f] $=_{def}$ stands for *equal by definition*.

2. $SEM_E(\epsilon) =_{def} \widehat{E}(\{\lambda\}) =_{def} \overline{E}(\lambda) =_{def} \Delta\mathcal{G}$.

3. $SEM_E(id) =_{def} \widehat{E}(\{id\}) =_{def} \overline{E}(id) =_{def} E(id)$.

4. To show this, we use the following statement which is shown by induction on the length of w_1. Let $w_1, w_2 \in ID^*$; then $\overline{E}(w_1 w_2) = \overline{E}(w_1) \circ \overline{E}(w_2)$.

   $\overline{E}(\lambda w_2) =_{def} \overline{E}(w_2) = \Delta\mathcal{G} \circ \overline{E}(w_2) =_{def} \overline{E}(\lambda) \circ \overline{E}(w_2)$. Assume that the statement holds for $w_1 \in ID^*$, and let $a \in ID$. Then $\overline{E}(aw_1w_2) =_{def} E(a) \circ \overline{E}(w_1w_2) =_{ind} E(a) \circ (\overline{E}(w_1) \circ \overline{E}(w_2)) = (E(a) \circ \overline{E}(w_1)) \circ \overline{E}(w_2) =_{def} \overline{E}(aw_1) \circ \overline{E}(w_2)$.[g] Now we get

   $$\begin{aligned} SEM_E(e_1 \,;\, e_2) &=_{def} \widehat{E}(L(e_1) \cdot L(e_2)) =_{def} \textstyle\bigcup_{w_1 \in L(e_1), w_2 \in L(e_2)} \overline{E}(w_1 w_2) \\ &= \textstyle\bigcup_{w_1 \in L(e_1), w_2 \in L(e_2)} \overline{E}(w_1) \circ \overline{E}(w_2) \\ &= \textstyle\bigcup_{w_1 \in L(e_1)} \overline{E}(w_1) \circ \bigcup_{w_2 \in L(e_2)} \overline{E}(w_2) =_{def} \widehat{E}(L(e_1)) \circ \widehat{E}(L(e_2)) \\ &=_{def} SEM_E(e_1) \circ SEM_E(e_2). \end{aligned}$$

5. $$\begin{aligned} SEM_E(e_1 \mid e_2) &=_{def} \widehat{E}(L(e_1) \cup L(e_2)) =_{def} \textstyle\bigcup_{w \in L(e_1) \cup L(e_2)} \overline{E}(w) \\ &= \textstyle\bigcup_{w_1 \in L(e_1)} \overline{E}(w_1) \cup \bigcup_{w_2 \in L(e_2)} \overline{E}(w_2) =_{def} \widehat{E}(L(e_1)) \cup \widehat{E}(L(e_2)) \\ &=_{def} SEM_E(e_1) \cup SEM_E(e_2). \end{aligned}$$

6. To show point 6, we first prove by induction on i that for $i \geq 0$, $\widehat{E}(L(e)^i) = SEM_E(e)^i$. If $i = 0$ we have $\widehat{E}(L(e)^0) =_{def} \widehat{E}(\{\lambda\}) =_{2.} \Delta\mathcal{G} =_{def} SEM_E(e)^0$. Moreover,

   $$\begin{aligned} \widehat{E}(L(e)^{i+1}) &=_{def} \widehat{E}(L(e) \cdot L(e)^i) =_{4.} \widehat{E}(L(e)) \circ \widehat{E}(L(e)^i) \\ &=_{ind} SEM_E(e) \circ SEM_E(e)^i =_{def} SEM_E(e)^{i+1}. \end{aligned}$$

   Hence,

   $$\begin{aligned} SEM_E(e^*) &=_{def} \widehat{E}(L(e)^*) =_{def} \widehat{E}(\textstyle\bigcup_{i=0}^{\infty} L(e)^i) = \bigcup_{i=0}^{\infty} \widehat{E}(L(e)^i) \\ &= \textstyle\bigcup_{i=0}^{\infty} SEM_E(e)^i =_{def} SEM_E(e)^*. \end{aligned}$$

□

1.2.6 Application Sequences

In interleaving sequences, rules are applied and imported transformation units are called in some order. Such sequences of applied rules and called transformation units help to clarify the role of control conditions of the language type as defined in 1.2.5.1.

[g] $=_{ind}$ stands for *equal by induction hypothesis.*

Let $trut = (I, U, R, C, T)$ be a transformation unit over some graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$. Assume that U and R are disjoint subsets of the set ID associated to $\mathcal{C}$. Then $x_1 \cdots x_n \in (U \cup R)^*$ $(x_i \in U \cup R)$ is called an *application sequence* of $(G, G') \in \mathcal{G} \times \mathcal{G}$ if there is an interleaving sequence $G_0, \ldots, G_n$ with $G_0 = G$, $G_n = G'$ and, for $i = 1, \ldots, n$, $G_{i-1} \Rightarrow_{x_i} G_i$ if $x_i \in R$ and $(G_{i-1}, G_i) \in SEM(x_i)$ if $x_i \in U$. In the case $n = 0$, the application sequence is the empty string λ.
Using these notions and notations, the following observation states that a language over $U \cup R$, used as a control condition due to 1.2.5.1, controls the order in which rules are applied and imported transformation units are actually used.

Observation 1.2.2
Let $\mathcal{C} = 2^{ID^*}$ be the class of control conditions of language type, and let $trut = (I, U, R, L, T)$ with $L \subseteq (U \cup R)^* \subseteq ID^*$. Then for all $G, G' \in \mathcal{G}$, the following statements are equivalent.

1. *$(G, G') \in SEM(trut)$.*
2. *$(G, G') \in SEM_{E(trut)}(L) \cap SEM(I) \times SEM(T)$.*
3. *There is an application sequence w of (G, G') with $w \in L$ and $(G, G') \in SEM(I) \times SEM(T)$.*

Proof
Let $(G, G') \in SEM(trut)$. Then by definition, there is an interleaving sequence in $trut$ from G to G', $(G, G') \in SEM_{E(trut)}(L)$, and $(G, G') \in SEM(I) \times SEM(T)$. Hence, point 1 implies point 2.
To show that point 2 implies point 3 and that point 3 implies point 1, we prove first the following claim:
$(G, G') \in SEM_{E(trut)}(L)$ iff there is an application sequence $w \in L$ of (G, G').
By definition, we have $(G, G') \in SEM_{E(trut)}(L) = \widehat{E(trut)}(L)$ iff $(G, G') \in \overline{E(trut)}(w)$ for some $w \in L$. We show now by induction on the structure of w that $(G, G') \in \overline{E(trut)}(w)$ iff w is an application sequence of (G, G').
If $w = \lambda$, we get $(G, G') \in \overline{E(trut)}(\lambda)$ iff $(G, G') \in \Delta\mathcal{G}$ iff $G = G'$ iff λ is an application sequence of (G, G').
Assume now that the statement holds for $v \in (U \cup R)^*$.
And consider $w = xv$ with $x \in U \cup R$. Then $(G, G') \in \overline{E(trut)}(xv) = E(trut)(x) \circ \overline{E(trut)}(v)$, means that there is some $\overline{G} \in \mathcal{G}$ with $(G, \overline{G}) \in E(trut)(x)$ and $(\overline{G}, G') \in \overline{E(trut)}(v)$. The latter implies by induction that v is an application sequence of $(\overline{G}, G')$ such that there is an interleaving sequence

$G_0, \ldots, G_n$ with $\overline{G} = G_0$ and $G' = G_n$. The former means $G \Rightarrow_x \overline{G}$ if $x \in R$ and $(G, \overline{G}) \in \mathit{SEM}(x)$ if $x \in U$. Altogether, $G, G_0, \ldots, G_n$ defines an interleaving sequence with xv as corresponding application sequence. Conversely, an application sequence xv of (G, G') is related to an interleaving sequence $G_0, \ldots, G_n$ with $G = G_0$, $G' = G_n$ and, in particular, $G_0 \Rightarrow_x G_1$ if $x \in R$ and $(G_0, G_1) \in \mathit{SEM}(x)$ if $x \in U$ such that $(G, G_1) \in E(\mathit{trut})(x)$ in any case. Moreover, v is an application sequence of (G_1, G_n) because $G_1, \ldots, G_n$ is an interleaving sequence. By induction hypothesis, we get $(G_1, G') \in \overline{E(\mathit{trut})}(v)$. The composition yields $(G, G') \in E(\mathit{trut})(x) \circ \overline{E(\mathit{trut})}(v) = \overline{E(\mathit{trut})}(xv)$. This completes the proof of the claim.
From the just proved claim follows directly that point 2 implies point 3.
Furthermore, let $w \in L$ be an application sequence of (G, G') with $(G, G') \in \mathit{SEM}(I) \times \mathit{SEM}(T)$. Then by definition there is an interleaving sequence in *trut* from G to G' with $(G, G') \in \mathit{SEM}(I) \times \mathit{SEM}(T)$, and by the claim, $(G, G') \in \mathit{SEM}_{E(\mathit{trut})}(L)$. Hence, $(G, G') \in \mathit{SEM}(\mathit{trut})$. This completes the proof. □

1.3 Shortest Paths—An Example

In this section, we specify the shortest-path algorithm of Floyd (cf. [34]) as a graph transformation unit to illustrate the usefulness of the concept. The algorithm is informally described in the next subsection. Its specification in form of a transformation unit is presented in subsection 1.3.3 after the underlying graph transformation approach is introduced. The rest of the section concerns the correctness and complexity of the specified algorithm. In subsection 1.3.4, we discuss the graphtheoretic background of the algorithm more formally to be able to prove correctness of the specification in subsection 1.3.5 (together with 1.3.7). Finally, the issue of complexity is studied in subsection 1.3.6.

1.3.1 Informal Description of the Algorithm

Given a directed graph G and a pair v and v' of nodes, the algorithm computes the distance of a shortest path from v to v' in G. The main idea is to compute the distance of a shortest path from v to v' that avoids certain nodes in intermediate steps from the distances of shortest paths that avoid more nodes. In more detail, the shortest paths from v to v' that avoid all nodes in intermediate steps are the edges from v to v' with minimum distance. Starting with these edges and distances, one may admit more and more nodes as intermediate ones. Whenever a new intermediate node $\overline{v}$ is admitted, one adds up the distance of the edge from v to $\overline{v}$ and the distance of the edge from $\overline{v}$ to v' (if there are any) and labels a new edge from v to v' with this sum. If there is already an edge, the one with the smaller distance is kept. The algorithm terminates after all nodes are admitted. Then an edge from v to v' is labelled with the distance of a shortest path from v to v' in the original graph, and there is no edge from v to v' if originally there is no path from v to v'. A formal treatment of the shortest-path algorithm is given in subsection 1.3.4.

1.3.2 Graph Transformation Approach

For the purposes of this illustration, a particular graph transformation approach is tailored. But the example is easily adapted to most of the general graph transformation approaches one encounters in the literature.

1. The class of graphs considered consists of directed graphs of the form $G = (V, E, s, t, l, \mathit{dist})$ where V is the set of *nodes*, E is the set of *edges*, $s : E \to V$ and $t : E \to V$ are mappings associating each edge $e \in E$ with a *source* $s(e)$ and a *target* $t(e)$, $l : V \to \{0, 1, 2\}$ is a node labelling and $\mathit{dist} : E \to \mathbb{N}$ is an edge labelling, called *distance*. Loops are forbidden, i.e. there is no edge e with $s(e) = t(e)$. The distance is essential, the node labelling serves only auxiliary purposes.

   Let $G = (V, E, s, t, l, \mathit{dist})$ and $G' = (V', E', s', t', l', \mathit{dist}')$ be two graphs. G is a *subgraph* of G' if $V \subseteq V', E \subseteq E'$ and $s(e) = s'(e)$, $t(e) = t'(e)$ and $\mathit{dist}(e) = \mathit{dist}'(e)$ for all $e \in E$ as well as $l(v) = l'(v)$ for all $v \in V$. G and G' are *isomorphic* if there are bijective mappings $f : V \to V'$ and $g : E \to E'$ with $f(s(e)) = s'(g(e))$, $f(t(e)) = t'(g(e))$ and $\mathit{dist}(e) = \mathit{dist}'(g(e))$ for all $e \in E$ as well as $l(v) = l'(f(v))$ for all $v \in V$.

2. The class of rules considered consists of pairs of graphs $r = (L, R)$ where L and R share the set of nodes. Moreover, the left-hand side L of r may be equipped with an extra context edge.

   Rules are presented in the form $L \longrightarrow R$ where an extra context edge in L is dashed. Nodes of L and R are drawn in the same fill style if they are equal. A node label is omitted if it is parametric, i.e. if it can be choosen arbitrarily before applying the rule.

3. A rule $r = (L, R)$ is applied to a graph G directly deriving G' in four steps: (1) look for a subgraph L_0 of G isomorphic to L, (2) remove the edges of L_0 from G, (3) add the edges of R (by using the isomorphism between L and L_0 to place the edges), and (4) change the node labels where they are different in L and R (again by using the isomorphism). If the left-hand side has got an extra context edge, this serves as a negative context condition meaning that the direct derivation is only admitted

if G does not contain such an edge between the respective nodes. This altogether is denoted by $G \Rightarrow_r G'$.

4. The only two used graph class expressions are the restrictions of the node label alphabet to $\{0\}$ and $\{2\}$ respectively. If the two expressions are denoted by *node label* i for $i \in \{0, 2\}$, *SEM*(*node label* i) contains all graphs the nodes of which are constantly labelled with i.

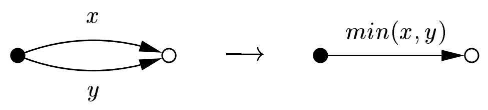
5. The used control conditions are regular expressions and *as long as possible* where the latter applies the rules of a transformation unit as long as possible (cf. 1.2.5.5).

In the following, transformation units are presented by indicating the components with respective keywords. Trivial components (i.e. no import, no rules, the graph class expression *all*, and the control condition *true*) are omitted.

1.3.3 Specification of the Algorithm

Floyd's algorithm is specified in terms of transformation units following the informal description of the algorithm. To be able to distinguish between forbidden nodes, a just admitted node and formerly admitted nodes, the node labels $0, 1$ and 2 resp. are used. Initially, all nodes carry the 0-label.
The main transformation unit *shortest-path* uses the transformation units *minimum*, *sum*, *change*$(0, 1)$ and *change*$(1, 2)$ in a certain order which is given by a regular control expression. The transformation unit *minimum* takes care of parallel edges. Its control condition *as long as possible* makes sure that all parallel edges are removed. The transformation unit *sum* sums up successive edges if their intermediate node is 1-labeled. Here the control condition *as long as possible* makes sure that all possible summations are performed, while the negative context condition associated to the rule prevents that the same summation is done twice. Finally, *change*(i, j) relabels a node from i to j. The control condition *once* guarantees that the rule of the transformation unit is applied exactly once in each of its derivations. The term *once* is a synonym for the rule considered as a regular expression. Note that in the transformation units *minimum* and *sum* the edge labels as well as most of the node labels of the rules are parametric, i.e. they can be choosen arbitrarily before each rule application.
Wherever parallel edges occur, only an edge with the minimum label is kept. This is achieved by the transformation unit *minimum*.

minimum
rules:

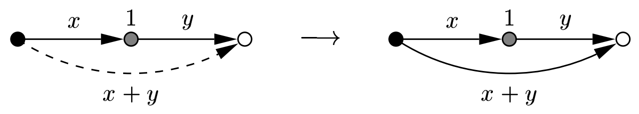


$(x, y \in \mathbb{N})$

conds: *as long as possible*

The summations of the distances of successive edges is performed by the transformation unit *sum* whenever the intermediate node is labelled with 1.

sum
rules:

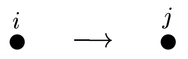


$(x, y \in \mathbb{N})$

conds: *as long as possible*

Before *sum* is applied, a node with label 0 gets label 1, and afterwards this label is changed into 2 by using the transformation unit *change*(i, j) for any two labels i, j.

change(i, j)
rules:



conds: *once*

After *minimum* has done its job once, the described sequence of transformation units

$$change(0, 1), sum, change(1, 2), minimum$$

can be iterated until all nodes are admitted. This yields the shortest-path algorithm.

```
shortest-path
    initial:   node label 0
    uses:      change(0,1), change(1,2), sum, minimum
    conds:     minimum ; (change(0,1) ; sum ; change(1,2) ; minimum)*
    terminal:  node label 2
```

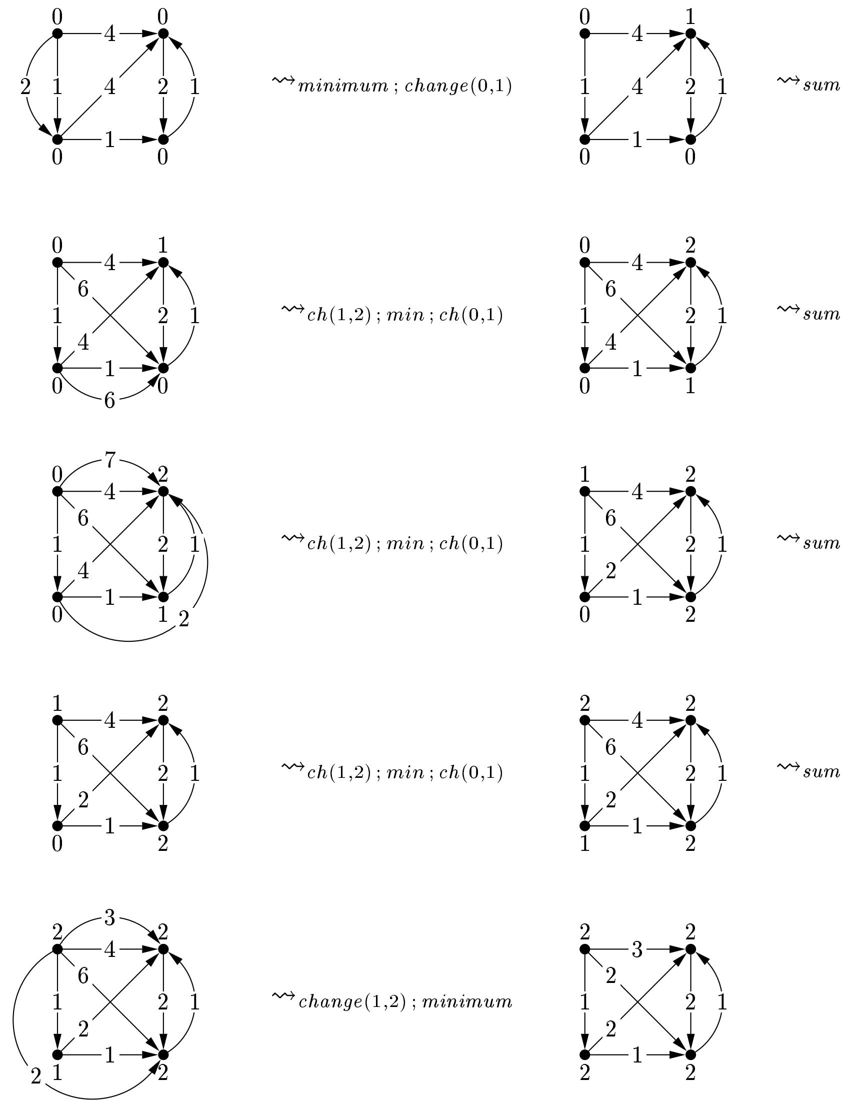
Figure 1.1 shows an interleaving sequence of *shortest-path* where the effect of the *change* units are not presented separately, but composed with *minimum*. Note that the regular expression $ch(1,2)\,;\, min\,;\, ch(0,1)$ in the picture is an abbreviation for the control condition $change(1,2)\,;\, minimum\,;\, change(0,1)$.

1.3.4 Graphtheoretic Background

To facilitate the correctness proof for the algorithm, the graphtheoretic background is needed explicitly.
Let (V, E, s, t) be an unlabelled directed graph without loops, and let $dist : E \to \mathbb{N}$ be a distance function on the edges.
A sequence of edges $p = e_1 \cdots e_n$ $(n \geq 1)$, is a *path from* v to v' with $dist(p) = \sum_{i=1}^{n} dist(e_i)$ if $s(e_1) = v$, $t(e_n) = v'$ and $t(e_i) = s(e_{i+1})$ for $i = 1, \ldots, n-1$. The set of the latter nodes $\{t(e_1), \ldots, t(e_{n-1}\}$, called *intermediate nodes*, is denoted by $inter(p)$. Moreover, for $v \in V$, the empty sequence λ is considered as a path from v to v with $dist(\lambda) = 0$ and $inter(\lambda) = \emptyset$. For $U \subseteq V$, a path p is said to *avoid* U if $inter(p) \cap U = \emptyset$.
Let $PATH(v, v', U)$ denote the set of all paths from v to v' avoiding U. If $PATH(v, v', U) \neq \emptyset$, the minimum distance is denoted by $short(v, v', U)$, i.e.

$$short(v, v', U) = min\{dist(p) \mid p \in PATH(v, v', U)\}.$$

A path $p_0 \in PATH(v, v', U)$ with $dist(p_0) = short(v, v', U)$ is called *shortest path* from v to v' *avoiding* U. (Note that $short(v, v', U)$ is undefined if $PATH(v, v', U) = \emptyset$.)
$PATH(v, v', \emptyset)$ consists of all paths from v to v' such that $short(v, v', \emptyset)$ is the minimum distance of all paths from v to v'. $PATH(v, v, U)$ contains the empty



$\rightsquigarrow_{minimum\,;\,change(0,1)}$
$\rightsquigarrow_{sum}$
$\rightsquigarrow_{ch(1,2)\,;\,min\,;\,ch(0,1)}$
$\rightsquigarrow_{sum}$
$\rightsquigarrow_{ch(1,2)\,;\,min\,;\,ch(0,1)}$
$\rightsquigarrow_{sum}$
$\rightsquigarrow_{ch(1,2)\,;\,min\,;\,ch(0,1)}$
$\rightsquigarrow_{sum}$
$\rightsquigarrow_{change(1,2)\,;\,minimum}$

Figure 1.1: Example of an interleaving sequence in *shortest_path*

path λ such that $short(v,v,U) = 0$ for all $v \in V$ and $U \subseteq V$. $PATH(v,v',V)$ with $v \neq v'$ consists of all edges e with $s(e) = v$ and $t(e) = v'$. If this set of edges is denoted by $E(v,v')$, one has

$$short(v,v',V) = min\{dist(e) \mid e \in E(v,v')\}$$

for $v \neq v'$ and $E(v,v') \neq \emptyset$.
Let $PATH(v,v',U) \neq \emptyset$ with $v \neq v'$ and $\overline{U} = U \cup \{\overline{v}\}$ for some $\overline{v} \in V - U$. Then the following hold.

(1) If $PATH(v,v',\overline{U}) \neq \emptyset$, then

$$short(v,v',U) \leq short(v,v',\overline{U})$$

because $PATH(v,v',\overline{U}) \subseteq PATH(v,v',U)$.

(2) If $PATH(v,\overline{v},\overline{U}) \neq \emptyset \neq PATH(\overline{v},v',\overline{U})$, then

$$short(v,v',U) \leq short(v,\overline{v},\overline{U}) + short(\overline{v},v',\overline{U})$$

because the composition of a path from v to $\overline{v}$ avoiding $\overline{U}$ with a path from $\overline{v}$ to v' avoiding $\overline{U}$ yields a path from v to v' avoiding U.

(3) If $p_0 = e_1 \cdots e_n$ is a shortest path from v to v' avoiding U and $\overline{v} \notin inter(p_0)$, then $p_0 \in PATH(v,v',\overline{U})$, and hence

$$short(v,v',\overline{U}) \leq dist(p_0) = short(v,v',U).$$

(4) Otherwise $\overline{v} \in inter(p_0)$. In this case, there are indices i and j such that $e_1 \cdots e_i \in PATH(v,\overline{v},\overline{U})$ and $e_j \cdots e_n \in PATH(\overline{v},v',\overline{U})$ where e_i is the first edge on p_0 entering $\overline{v}$ and e_j is the last one leaving $\overline{v}$. Hence $i \leq j$ and

$$\begin{aligned} short(v,\overline{v},\overline{U}) + short(\overline{v},v',\overline{U}) &\leq dist(e_1 \cdots e_i) + dist(e_j \cdots e_n) \\ &\leq dist(p_0) \\ &= short(v,v',U). \end{aligned}$$

(5) Altogether, one gets either

$$short(v,v',U) = short(v,v',\overline{U})$$

or

$$short(v,v',U) = short(v,\overline{v},\overline{U}) + short(\overline{v},v',\overline{U}).$$

(6) And if both right-hand sides are defined, one has

$$\mathit{short}(v,v',U) = \mathit{min}\{\mathit{short}(v,v',\overline{U}), \mathit{short}(v,\overline{v},\overline{U}) + \mathit{short}(\overline{v},v',\overline{U})\}.$$

1.3.5 Correctness

In this section, the correctness of the transformation unit *shortest-path* is shown with respect to the function *short*. To achieve this, the interleaving semantics of all involved transformation units are characterized in graph-theoretic terms. Given a graph $G = (V, E, s, t, l, \mathit{dist})$, its components may be denoted by V_G, E_G, s_G, t_G, l_G and dist_G respectively.

1. The interleaving sequences of $\mathit{change}(i, j)$ are the direct derivations in which the only rule is applied to an arbitrary graph because nothing is imported and there is no restriction of initial and terminal graphs, but exactly one rule application is allowed. If the rule is applied, a node with label i gets label j obviously. Hence one gets:

   $(G, H) \in \mathit{SEM}(\mathit{change}(i, j))$ iff $V_G = V_H, E_G = E_H$, $s_G = s_H$, $t_G = t_H$, $\mathit{dist}_G = \mathit{dist}_H$ and there is some $v_0 \in V_G$ with $l_G(v_0) = i$, $l_H(v_0) = j$ and $l_G(v) = l_H(v)$ for all $v \neq v_0$.

2. If the rule of the transformation unit *minimum* is applied, two parallel edges are replaced by one edge labelled with the minimum distance of the replaced edges. The control condition makes sure that this is iterated as long as possible such that resulting graphs are simple (i.e. without parallel edges). This leads to the following characterization of the interleaving semantics of *minimum*.

   $(G, H) \in \mathit{SEM}(\mathit{minimum})$ iff $V_G = V_H$, $l_G = l_H$, and for all $v, v' \in V_G$ $E_G(v, v') \neq \emptyset$ iff $|E_H(v, v')| = 1$ and $\mathit{dist}_H(e_0) = \mathit{min}\{\mathit{dist}_G(e) \mid e \in E_G(v, v')\}$, where $\{e_0\} = E_H(v, v')$.[h]

3. If the rule of the transformation unit *sum* is applied, a new edge from a node v to a node v' is added with the label $x + y$ provided that there is not already such an edge and there are an x-labelled edge from v some node $\overline{v}$ (with label 1) and a y-labelled edge from $\overline{v}$ to v'. Due to the control condition this is iterated as long as possible. In other words, one gets the following characterization of the interleaving semantics of *sum*.

[h] $|E_H(v, v')|$ denotes the cardinality of $E_H(v, v')$.

$(G,H) \in \mathit{SEM}(\mathit{sum})$ iff $V_G = V_H$, $l_G = l_H$, $E_G \subseteq E_H$, $s_G = s_H|_{E_G}$, $t_G = t_H|_{E_G}$, $\mathit{dist}_G = \mathit{dist}_H|_{E_G}$, and for all $v, v' \in V_G$ $e_0 \in E_H(v,v') - E_G(v,v')$ with $\mathit{dist}_H(e_0) = c$ iff there are $\overline{v} \in V_G$ with $l_G(\overline{v}) = 1$, $e \in E_G(v,\overline{v})$, $e' \in E_G(\overline{v},v')$ with $\mathit{dist}_G(e) + \mathit{dist}_G(e') = c$ and there is no $\overline{e} \in E_G(v,v')$ with $\mathit{dist}_G(\overline{e}) = c$.[i]

4. $(G,H) \in \mathit{SEM}(\textit{shortest-path})$ means according to the control condition that there is an interleaving sequence of the form $G, G_0, \ldots, G_{4n}$ for some $n \in \mathbb{N}$ with $G_{4n} = H$ and

$$\begin{array}{ll} (1) & (G,G_0) \in \mathit{SEM}(\mathit{minimum}), \\ (2i) & (G_{4i},G_{4i+1}) \in \mathit{SEM}(\mathit{change}(0,1)) \\ (3i) & (G_{4i+1},G_{4i+2}) \in \mathit{SEM}(\mathit{sum}) \\ (4i) & (G_{4i+2},G_{4i+3}) \in \mathit{SEM}(\mathit{change}(1,2)) \\ (5i) & (G_{4i+3},G_{4i+4}) \in \mathit{SEM}(\mathit{minimum}) \end{array} \left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \text{for } i = 0,\ldots,n-1.$$

Due to 1. to 3., the set of nodes is invariant.

G is an initial graph meaning that $l_G(v) = 0$ for all $v \in V_G$. Using again the characterizations in 1. to 3., one gets $l_G = l_{G_0}$, and, for $i = 0,\ldots,n-1$, $l_{G_{4i+1}} = l_{G_{4i+2}}$, $l_{G_{4i+3}} = l_{G_{4i+4}}$. Moreover there is a node $v_{i+1} \in V_{G_{4i}}$ with $l_{G_{4i}}(v_{i+1}) = 0$, $l_{G_{4i+1}}(v_{i+1}) = 1$, $l_{G_{4i+3}}(v_{i+1}) = 2$ as well as $l_{G_{4i}}(v) = l_{G_{4i+1}}(v)$ and $l_{G_{4i+2}}(v) = l_{G_{4i+3}}(v)$ for all $v \neq v_{i+1}$. Moreover, $G_{4n} = H$ is a terminal graph such that $l_{G_{4n}}(v) = 2$ for all $v \in V_{G_{4n}} = V_H = V_G$. Hence n is the number of nodes because a single node is relabeled for each $i = 0,\ldots,n-1$.

Let $V_0 = \emptyset$ and $V_j = \{v_1,\ldots,v_j\}$ for $j = 1,\ldots,n$. Then the following holds for $j = 0,\ldots,n$, for $v,v' \in V_{G_{4j}}$ with $v \neq v'$ and for some $c \in \mathbb{N}$:

$$e_0 \in E_{G_{4j}}(v,v') \text{ with } \mathit{dist}_{G_{4j}}(e_0) = c \text{ iff } \mathit{short}(v,v',V_G - V_j) = c.$$

The proof of this statement can be found in subsection 1.3.7.

For $n = j$, this statement provides a characterization of the interleaving semantics of the transformation unit *shortest-path*.

$(G,H) \in \mathit{SEM}(\textit{shortest-path})$ iff $V_G = V_H$, $l_G(v) = 0$ and $l_H(v) = 2$ for all $v \in V_G$, H is simple, and, for all $v,v' \in V_G$ with $v \neq v'$ and for some $c \in \mathbb{N}$, there is some $e_0 \in E_H(v,v')$ with $\mathit{dist}_H(e_0) = c$ iff $\mathit{short}(v,v',\emptyset) = c$.

[i]For a function $f\colon A \to B$ and a set $C \subseteq A$ the restriction of f to C is denoted by $f|_C$.

In other words, *shortest-path* computes the distances of shortest paths in G.

1.3.6 Complexity

The length of a derivation reflects the complexity of the process to a certain degree because its products with lower and upper bounds for the cost of a direct derivation give corresponding bounds for the derivation. The length of derivations can be generalized to interleaving sequences (keeping in mind that derivations of different lengths may yield the same pair of graphs in the interleaving semantics).
Let $(G, H) \in \mathit{SEM}(t_0)$ for some transformation unit without import. Then $u(G, H)$ denotes the least upper bound of lengths of derivations from G to H (provided that the upper bound exists).
Let $s = G_0, \ldots, G_n$ be an interleaving sequence in the transformation unit t with the import units $t_1, \ldots, t_k$. Let, for $i = 1, \ldots, n$, $\mathit{val}(G_{i-1}, G_i) = u(G_{i-1}, G_i)$ if $(G_{i-1}, G_i) \in \mathit{SEM}(t_{j_i})$ for some $j_i \in \{1, \ldots, k\}$ and $\mathit{val}(G_{i-1}, G_i) = 1$ otherwise. Then $\mathit{val}(s) = \sum_{i=1}^{n} \mathit{val}(G_{i-1}, G_i)$ is called *value* of s.
Let $(G, H) \in \mathit{SEM}(t)$. Then $u(G, H)$ denotes the least upper bound of values of interleaving sequences from G to H in t (provided that the upper bound exists).
Finally, let G be some initial graph of t. Then $u(G)$, called the *upper length bound of* G, denotes the least upper bound of the $u(G, H)$ for all graphs H with $(G, H) \in \mathit{SEM}(t)$ provided that such a graph and the bound exist.
Now the upper length bound for the initial graphs of *shortest-path* is presented. Let G be such a graph, n its number of nodes and m its *multiplicity*, i.e. the smallest number of edges one has to remove to transform G into a simple graph. Then $u(G)$ is of the order $max(m, n^3)$.
This can be seen as follows. Consider the interleaving sequence in 4. (there is always one, and all have this form). Then on gets: (1) $u(G, G_0) = m$ because each application of the rule of *minimum* decreases the multiplicity by 1.
(2i) $u(G_{4i}, G_{4i+1}) = 1$ because of the control condition of *change*$(0, 1)$.
(3i) $u(G_{4i+1}, G_{4i+2}) \leq (n-1) \cdot (n-2)$ because v_{i+1} is the only 1-labelled node which may be adjacent to any of the $(n-1) \cdot (n-2)$ pairs of other nodes. Hence there may be as many occurrences for the rule of *sum*. But no occurrence can be used twice because of the negative application condition.
(4i) $u(G_{4i+2}, G_{4i+3}) = 1$ because of the control condition of *change*$(1, 2)$.
(5i) $u(G_{4i+3}, G_{4i+4}) \leq (n-1) \cdot (n-2)$ because the derivation in (3i) produces up

to $(n-1)\cdot(n-2)$ new edges that may be parallel to old ones. Hence *minimum* takes up to $(n-1)\cdot(n-2)$ rule applications to get rid of this multiplicity. Altogether, this amounts to

$$\begin{aligned} u(G,H) &\leq m + \sum_{i=0}^{n-1}(1+(n-1)(n-2)+1+(n-1)(n-2)) \\ &= m + 2n + 2n(n-1)(n-2) \end{aligned}$$

for all H with $(G,H) \in \mathit{SEM}(\mathit{shortest\text{-}path})$ proving the statement for $u(G)$. The upper length bounds for *shortest-path* are in the order of upper bounds for the computational costs because all involved derivations can be organized in such a way that the cost of a direct derivation is constant. For this purpose, one needs direct access to the nodes making the steps (2i) and (4i) constant as well as to the pairs of nodes making the direct derivations in the other steps constant. This means that the transformation unit specification of Floyd's algorithm is exactly as efficient as the versions of the algorithm in the literature.

1.3.7 Completing the Proof

This subsection provides the proof of the statement of 1.3.5.4. It is done by induction on j.
For $j = 0$, $e_0 \in E_{G_0}(v,v')$ means due to 2. and subsection 1.3.4 that $\mathit{dist}_{G_0}(e_0) = \mathit{min}\{\mathit{dist}_G(e) \mid e \in E_G(v,v')\} = \mathit{short}(v,v',V_G) = \mathit{short}(v,v',V_G - V_0)$.
Assume that the statement holds for some $j \geq 0$.
Consider now $j+1$. If $e_0 \in E_{G_{4j+4}}(v,v')$, then either (1) $e_0 \in E_{G_{4j+3}}(v,v')$ or (2) there are $e_1, e_2 \in E_{G_{4j+3}}(v,v')$ with $\mathit{dist}_{G_{4j+4}}(e_0) = \mathit{min}\{\mathit{dist}_{G_{4j+3}}(e_1), \mathit{dist}_{G_{4j+3}}(e_2)\}$ (cf. 2.). Moreover, $E_{G_{4j+3}} = E_{G_{4j+2}}$.
In the first case, either (1.1) $e_0 \in E_{G_{4j+1}}(v,v')$ or (1.2) there are $e_3 \in E_{G_{4j+1}}(v,v_{j+1})$ and $e_4 \in E_{G_{4j+1}}(v_{j+1},v')$ with $\mathit{dist}_{G_{4j+2}}(e_0) = \mathit{dist}_{G_{4j+1}}(e_3) + \mathit{dist}_{G_{4j+1}}(e_4)$ (cf. 3.).
In the second case, one of the edges say e_1, belongs to G_{4j+1}, and the other edge is constructed by the application of the *sum*-rule because G_{4j} is a simple graph as a result of *minimum*, $G_{4_{j+1}}$ is simple as a relabelling of G_{4j} and *sum* adds at most one edge per pair of nodes. This means $e_1 \in E_{G_{4j+1}}(v,v')$ and there are $e_3 \in E_{G_{4j+1}}(v,v_{j+1})$ and $e_4 \in E_{G_{4j+1}}(v_{j+1},v')$ with $\mathit{dist}_{G_{4j+2}}(e_2) = \mathit{dist}_{G_{4j+1}}(e_3) + \mathit{dist}_{G_{4j+1}}(e_4)$.
Moreover, $E_{G_{4j+1}}(v,v') = E_{G_{4j}}$.
Therefore the induction hypothesis can be applied in all cases yielding the following:
(1.1) $\mathit{dist}_{G_{4j}}(e_0) = \mathit{short}(v,v',V_G - V_j)$,

(1.2) $dist_{G_{4j}}(e_3) = short(v, v_{j+1}, V_G - V_j)$ and $dist_{G_{4j}}(e_4) = short(v_{j+1}, v', V_G - V_j)$,
(2) $dist_{G_{4j}}(e_1) = short(v, v', V_G - V_j)$, $dist_{G_{4j}}(e_3) = short(v, v_{j+1}, V_G - V_j)$ and $dist_{G_{4j}}(e_4) = short(v_{j+1}, v', V_G - V_j)$.
If $E_{G_{4j}}(v, v_{j+1}) = \emptyset$ or $E_{G_{4j}}(v_{j+1}, v') = \emptyset$, then $short(v, v_{j+1}, V_G - V_j)$ or $short(v_{j+1}, v', V_G - V_j)$ is undefined (otherwise there would be an edge from v to v_{j+1} and v_{j+1} to v'). Hence $short(v, v', V_G - V_{j+1}) = short(v, v', V_G - V_j)$ as shown in subsection 1.3.4. This yields together with the other parts of case (1.1):

$$dist_{G_{4j+4}}(e_0) = dist_{G_{4j}}(e_0) = short(v, v', V_G - V_j) = short(v, v', V_G - V_{j+1}).$$

This remains true even if $E_{G_{4j}}(v, v_{j+1}) \neq \emptyset \neq E_{G_{4j}}(v_{j+1}, v')$ because $dist_{G_{4j}}(e_0) = dist_{G_{4j}}(e_3) + dist_{G_{4j}}(e_4)$ for $e_3 \in E_{G_{4j}}(v, v_{j+1})$ and $e_4 \in E_{G_{4j}}(v_{j+1}, v')$ otherwise the *sum*-step (3j) would produce a new edge from v to v' which is either case (1.2) or (2)).
In case (1.2), $E_{G_{4j}}(v, v') = \emptyset$ because otherwise one would be faced with case (2). Using the result in subsection 1.3.4 (together with the reasoning in (1.2)), one gets:

$$\begin{aligned} dist_{G_{4j+4}}(e_0) &= dist_{G_{4j}}(e_3) + dist_{G_{4j}}(e_4) \\ &= short(v, v_{j+1}, V_G - V_j) + short(v_{j+1}, v', V_G - V_j), \\ &= short(v, v', V_G - V_{j+1}). \end{aligned}$$

It remains case (2) where one gets analogously:

$$\begin{aligned} dist_{G_{4j+4}}(e_0) &= min\{dist_{G_{4j+3}}(e_1), dist_{G_{4j+3}}(e_2)\} \\ &= min\{dist_{G_{4j}}(e_1), dist_{G_{4j}}(e_3) + dist_{G_{4j}}(e_4)\} \\ &= min\{short(v, v', V_G - V_j), \\ &\quad short(v, v_{j+1}, V_G - V_j) + short(v_{j+1}, v', V_G - V_j)\} \\ &= short(v, v', V_G - V_{j+1}). \end{aligned}$$

Conversely, let $short(v, v', V_G - V_{j+1}) = c$. Then there is a $p_0 \in PATH(v, v', V_G - V_{j+1})$ with $dist_G(p_0) = c$. If $v_{j+1} \notin inter(p_0)$, then $p_0 \in PATH(v, v', V_G - V_j)$ and hence $short(v, v', V_G - V_j) = short(v, v', V_G - V_{j+1})$ (cf. subsection 1.3.4). By induction hypothesis, one gets $e_0 \in E_{G_{4j}}(v, v')$ with $dist_{G_{4j}}(e_0) = c$. If e_0 is still in G_{4j+4}, one is done. If e_0 is not in G_{4j+4}, then the following happens: There are $e_1 \in E_{G_{4j+1}}(v, v_{j+1})$ and $e_2 \in E_{G_{4j+1}}(v_{j+1}, v')$ with $d = dist_{G_{4j+1}}(e_1) + dist_{G_{4j+1}}(e_2) \neq c$ such that *sum* in step (3j) produces $e_4 \in E_{G_{4j+2}}(v, v')$ with $dist_{G_{4j+2}}(e_4) = d$. Therefore, e_0 and e_4 are replaced by *minimum* in step (5j) by $e_5 \in E_{G_{4j+4}}(v, v')$ with $dist_{G_{4j+4}}(e_5) = min\{c, d\}$.

By induction hypothesis and the results in subsection 1.3.4, one gets

$$\begin{aligned} d &= \mathit{short}(v, v_{j+1}, V_G - V_j) + \mathit{short}(v_{j+1}, v', V_G - V_j) \\ &\geq \mathit{short}(v, v', V_G - V_{j+1}) = c \end{aligned}$$

such that one is done again.
It remains the case $v_{j+1} \in \mathit{inter}(p_0)$. Using the observations in subsection 1.3.4, one gets

$$\mathit{short}(v, v_{j+1}, V_G - V_j) + \mathit{short}(v_{j+1}, v', V_G - V_j) = c.$$

By induction hypothesis, there are $e_1 \in E_{G_{4j}}(v, v_{j+1})$ with $d_1 = \mathit{dist}_{G_{4j}}(e_1) = \mathit{short}(v, v_{j+1}, V_G - V_j)$ and $e_2 \in E_{G_{4j}}(v_{j+1}, v')$ with $d_2 = \mathit{dist}_{G_{4j}}(e_2) = \mathit{short}(v_{j+1}, v', V_G - V_j)$.
After step (3j), this guarantees some $e_3 \in E_{G_{4j+2}}(v, v')$ with $\mathit{dist}_{G_{4j+2}}(e_3) = d_1 + d_2 = c$. If e_3 is still in G_{4j+4}, one is done again. Otherwise there is $e_0 \in E_{G_{4j}}(v, v')$ with $d = \mathit{dist}_{G_{4j}}(e_0) \neq c$, and *minimum* produces $e_5 \in E_{G_{4j+4}}(v, v')$ with $\mathit{dist}_{G_{4j+4}}(e_5) = \mathit{min}\{c, d\}$ in step (5j). By induction hypothesis, and the results in subsection 1.3.4, one gets

$$d = \mathit{short}(v, v', V_G - V_j) \geq \mathit{short}(v, v', V_G - V_{j+1}) = c$$

such that one is done also in this last case.
This completes the proof.

1.4 Transformation Modules

In the sample specification above, the transformation unit *shortest-path* is the unit of interest while all the others are of an auxiliary nature. Moreover, the shortest-path algorithm may be part of a route planning system providing further graph algorithms. Clearly, a specification language based on graph transformation should provide the means to put together several transformation units if they belong to the same application and to distinguish between main and auxiliary transformation units. This is accomplished by the notion of a transformation module that combines a set of transformation units. Some of them may be indicated as members of the export interface whereas the rest is hidden. In addition, there may be an import interface consisting of formal parameter units being transformation units of which only initial and terminal graphs are specified. This allows to leave parts of a system unspecified. They may be filled in later by instantiation.
The following notion of a module is a variant of the simple modules as proposed by Heckel et al. [35], who use a lightened form of units.

1.4.1 Formal Parameter Units and Modules

A transformation unit *formal* is a *formal parameter unit* if $U_{formal} = \emptyset$, $R_{formal} = \emptyset$, and $C_{formal} = \mathit{true}$. While the semantic relation of *formal* is empty, the binary relation $\mathit{SEM}(I_{formal}) \times \mathit{SEM}(T_{formal})$ describes the upper bound of any actual parameter unit *actual* subject to the condition $\mathit{SEM}(\mathit{actual}) \subseteq \mathit{SEM}(I_{formal}) \times \mathit{SEM}(T_{formal})$.
A *transformation module* is a triple $MOD = (IMPORT, BODY, EXPORT)$ where $IMPORT$ is a set of formal parameter units, $BODY$ is a set of transformation units each of which using only units from $BODY$ and $IMPORT$, i.e. $U_t \subseteq IMPORT \cup BODY$ for each $t \in BODY$, and $EXPORT$ is a subset of $BODY \cup IMPORT$.
Based on the interleaving semantics of transformation units, the semantics of a transformation module $MOD = (IMPORT, BODY, EXPORT)$ is easily defined. Choosing a relation $\mathit{SEM}(im) \subseteq \mathit{SEM}(I_{im}) \times \mathit{SEM}(T_{im})$ for each $im \in IMPORT$, $\mathit{SEM}(t)$ for each $t \in BODY - IMPORT$ is the interleaving semantics as defined level by level in the second section. Then $\mathit{SEM}(MOD)$ is just the restriction to the export interface, i.e. the family of relations $\mathit{SEM}(ex)$ for each $ex \in EXPORT$.

1.4.2 Composition of Modules

If one wants to instantiate some of the formal parameter units of a module, one can choose exported transformation units of another module as actual parameters. Let $MOD = (IMPORT, BODY, EXPORT)$ and $MOD' = (IMPORT', BODY', EXPORT')$ be two transformation modules and $a\colon IMPORT \to EXPORT'$ be a partial mapping assigning export units of MOD' to some import units of MOD. Then a composition $MOD \circ_a MOD'$ can be constructed by merging $IMPORT$ and $EXPORT'$ with respect to a. In more detail, the domain of definition dom_a of a is removed from $IMPORT$, $BODY$ and $EXPORT$ are actualized according to a, meaning that each $im \in dom_a$ is replaced by $a(im)$ wherever it occurs in $BODY$ and $EXPORT$. Denoting the results by $a(BODY)$ and $a(EXPORT)$ resp., everything else is kept as it is. This yields

$$\begin{aligned} MOD \circ_a MOD' = (\quad & (IMPORT - dom_a) \cup IMPORT', \\ & a(BODY) \cup BODY', \\ & a(EXPORT)). \end{aligned}$$

Alternatively, one could keep the export interface $EXPORT'$ in addition as export of the composition.

Obviously, the composition is again a transformation module such that one can get rid of all formal parameter units eventually by repeated composition.

1.4.3 Example

Floyd's algorithm as specified in the previous section can be represented as the transformation module

Floyd	
import:	—
body:	*shortest-path*, *minimum*, *sum*, $change(0,1)$, $change(1,2)$
export:	*shortest-path*

where the units *minimum*, *sum*, $change(0,1)$, and $change(1,2)$ are hidden and only the *shortest-path* unit is exported. Nothing is imported because the five units are completely defined by themselves.
As an illustration of the use of formal import parameters, consider the module **2-nodes** which imports the formal parameter unit

relabel	
initial:	*node label* 0
terminal:	*node label* 2

and the body of which consists of the three transformation units *pre*, *post*, and *A_to_B* where the latter is exported. The unit *pre* adds to an arbitrary initial graph two new nodes A and B, an edge from A to some old node and an edge from another old node to B. Moreover, the new edges get the distances 0 and all nodes are labelled with 0. The unit *post* removes all nodes except A and B and all edges except those connecting A and B. The explicit specifications of *pre* and *post* are omitted. Finally, the unit *A_to_B* is given by

A_to_B	
uses:	*pre*, *relabel*, *post*
conds:	*pre* ; *relabel* ; *post*

The module **2-nodes** may be seen as a kind of query concerning two nodes of a graph depending on the actual choice for *relabel*. Let $SEM(relabel)$ be some relation between graphs with 0-labeled nodes and graphs with 2-labeled

nodes. Then the semantic relation of *A_to_B*, which is also the semantics of the module takes an arbitrary graph, distinguishes two nodes through the call of *pre*, applies *SEM*(*relabel*) and restricts the result to the distiguished nodes (as far as they are still present). If one chooses particularly *SEM*(*shortest-path*), one gets the distance of the shortest path between the two distinguished nodes. Clearly, other actual parameters yield other answers.
The instantiation of *relabel* by *shortest-path* can be done explicitly by composition

2-nodes ∘ Floyd	
import:	—
body:	*pre*, *post*, *A_to_B*, *shortest-path*, *minimum*, *sum*, *change*$(0,1)$, *change*$(1,2)$
export:	*A_to_B*

where the actualized form of *A_to_B* is

A_to_B	
uses:	*pre*, *shortest-path*, *post*
conds:	*pre* ; *shortest-path* ; *post*

All other units are not affected by the instantiation. The actualization mapping is omitted because *relabel* is the only formal parameter and *shortest-path* the only exported unit that can be assigned.

1.5 Conclusion

In this chapter, we have introduced and illustrated the syntactic and semantic features of transformation units and transformation modules as structuring concepts for graph transformation systems. We have restricted ourselves to the case of acyclic use relations. The more complicated case of networks of transformation units which may use each other arbitrarily is studied in Kreowski, Kuske, and Schürr [16]. In this case, the semantic relations of the units in the network are aggregated by iterated interleaving semantics and yield a fixed point semantics under suitable assumptions on the control conditions. Moreover, the reader can find an investigation of operations on transformation units and corresponding operations on semantic relations in [14].
Transformation units and modules are the main structuring concepts of the rule and graph centered specification language GRACE (see [6, 14, 15]) which

is not based on a particular graph transformation approach as other graph transformation languages. GRACE is planned as a visual language with a graphical interface that supports the visualized edition of graphs and rules and the animated execution of interleaving sequences. Moreover, GRACE will provide an explanation component supporting structured correctness of proofs in particular. There is a very first experimental implementation of GRACE available: GRACEland provides a 3D editor and interpreter for the GRACE language including transformation units and modules, but is not yet approach-independent. The system is part of Martin Faust's diploma thesis (see http://www.informatik.uni-bremen.de/grp/ag-ti/GRACEland for more details).

In chapter **??**, transformation units and (a variant of) transformation modules are compared with other module concepts within the framework of graph transformation in various respects. It should be noted that most other notions are based on particular graph transformation approaches. Moreover, the comparison does not concern the semantic level explicitly while in our presentation the semantics is essential.

Although transformation units and transformation modules seem to work very well, they do not cover all aspects and needs of structuring. First, they allow to structure the set of rules of a graph transformation system, but the graphs are kept as whole entities which may get very large. A compatible structuring concept for cutting large graphs into small pieces is missing. Second, the underlying graph transformation approach may provide parallel rules and in this way parallel applications of rules. In all other respects, the interleaving semantics is sequential and does not describe concurrent system activities. Third, the interleaving semantics yields binary relations on graphs. This includes graph language generation and recognition if one fixes the first or second component of the relation. But what about n-ary relations on graphs with $n > 2$ or relations involving other data types than graphs? Fourth, the introduced notions are approach-independent. But all transformation units within a module are based on the same approach. The interconnection and interaction of units and modules would become more sophisticated if one would allow the coexistence of various approaches within a module supporting the switch from directed to undirected graphs for example. Fifth, a specification in terms of transformation modules describes the sytem on a fixed level of abstraction. A compatible notion of refinement is needed if one wants to get rid of this restriction (cf. [36]). There is still a lot to do. Future research will fill these and further deficiencies in one way or other.

References

1. Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, 1979.
2. Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, 1983.
3. Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors. *Proc. Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, 1987.
4. Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, 1991.
5. Hans-Jürgen Schneider and Hartmut Ehrig, editors. *Proc. Graph Transformations in Computer Science*, number 776 in Lecture Notes in Computer Science, 1994.
6. Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 1998. To appear. Also as technical report no 7/96, Universität Bremen.
7. Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, 1996.
8. Andy Schürr. PROGRES: A VHL-language based on graph grammars. In Ehrig et al. [4], pages 641–659.
9. Andy Schürr, Andreas Winter, and Albert Zündorf. PROGRES: Language and environment. 1999. This volume.
10. Michael Himsolt. Graph$^{\mathrm{ed}}$: An interactive tool for developing graph grammars. In Ehrig et al. [4], pages 61–65.
11. John R. W. Glauert, J. Richard Kennaway, and M. Ronan Sleep. DACTL: An experimental graph rewriting language. In Ehrig et al. [4], pages 378–395.
12. Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In Claude Kirchner, editor, *Proc. Rewriting Techniques and Applications*, number 690 in Lecture Notes in Computer Science, pages 451–456, 1993.
13. Gabriele Taentzer, C. Ermel, and Michael Rudolf. The AGG approach: Language and environment. 1999. This volume.

14. Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units — a step into GRACE. In Cuny et al. [7], pages 89–108.
15. Andy Schürr. Programmed graph transformations and graph transformation units in GRACE. In Cuny et al. [7], pages 122–136.
16. Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1998.
17. Horst Bunke. Programmed graph grammars. In Claus et al. [1], pages 155–166.
18. Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierungen*. Vieweg, Braunschweig, 1979.
19. Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In Grzegorz Rozenberg and Arto Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, Berlin, 1986.
20. Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars, I and II. *Information Sciences*, 52:185–210 and 221–246, 1990.
21. Andrea Maggiolo-Schettini and Józef Winkowski. A programming language for deriving hypergraphs. volume 581 of *Lecture Notes in Computer Science*, pages 221–231, 1992.
22. Andy Schürr and Albert Zündorf. Nondeterministic control structures for graph rewriting systems. In G. Schmidt and Rudolf Berghammer, editors, *Proc. Graph-Theoretic Concepts in Computer Science*, number 570 in Lecture Notes in Computer Science, pages 48–62, 1991.
23. Hans-Jörg Kreowski. Five facets of hyperedge replacement beyond context-freeness. In Z. Ésik, editor, *Proc. Fundamentals of Computation Theory*, number 710 in Lecture Notes in Computer Science, pages 69–86, 1993.
24. Igor Litovsky and Yves Métivier. Computing with graph rewriting systems with priorities. *Theoretical Computer Science*, 115:191–224, 1993.
25. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, XXVII:1/2, 1996.
26. Andrea Maggiolo-Schettini and Józef Winkowski. A kernel language for programmed rewriting of (hyper)graphs. *Acta Informatica*, 33(6):523–546, 1996.
27. Andy Schürr. Programmed graph replacement systems. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*, pages 479–546. World Scientific, Singapore, 1997.

28. Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory.* Number 18 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1989.
29. Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [1], pages 1–69.
30. Dirk Janssens and Grzegorz Rozenberg. On the structure of node-label-controlled graph languages. *Information Sciences*, 20:191–216, 1980.
31. Bruno Courcelle. Graph rewriting: An algebraic and logical approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 193–242. Elsevier, Amsterdam, 1990.
32. Annegret Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.
33. Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
34. Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.
35. Reiko Heckel, Berthold Hoffmann, Peter Knirsch, and Sabine Kuske. Simple modules for GRACE. In *Preliminary Proc. Theory and Application of Graph Transformations*, 1998. To appear.
36. Martin Grosse-Rhode, Francesco Parisi Presicce, and Marta Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Mathematical Foundations of Computer Science*, volume 1450, pages 553–561, 1998.