Universität der Bundeswehr München

A Generic Approach to the Recognition and Analysis of Sketched Diagrams Using Context Information

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inf. Florian Brieler
am 25. Juni 2009

Vorsitzender der Kommission:	Prof. Dr. Peter Hertling
Betreuer und 1. Berichterstatter:	Prof. Dr.-Ing. Mark Minas
2. Berichterstatter:	Prof. Gennaro Costagliola
1. Prüfer:	Prof. Dr. Gunnar Teege
2. Prüfer:	Prof. Klaus Buchenrieder, Ph.D.
Tag der mündlichen Prüfung:	08. Februar 2010

Universität der Bundeswehr München
Fakultät für Informatik

Abstract

Recent decades have shown the rise of diagrammatic representation of information. In computer sciences, for example, general purpose diagrammatic notations like the UML are an everyday tool nowadays, and can even be considered as common knowledge. Also the advent of domain-specific languages (DSLs) can be observed. On the other hand, the research field of sketching is becoming popular, due to advances in processing speed and input hardware. Also, fields of application are evident, e.g. drawing of diagrams. The term sketching means to have a user draw something and have the computer interpret the drawing in some appropriate way. The advantage of sketching over traditional WIMP-based user interfaces (window, icon, menu, pointing device) is a more natural and intuitive way of interaction with the computer.

This thesis presents DSKETCH, an approach to sketching of diagrams. The idea is that the user first draws a diagram, and then DSKETCH derives the syntactic and semantic information conveyed in the drawing. The semantic information can be used for subsequent processing. The approach is fully generic, i.e., it is not tailored to a specific diagram language. There is a prototypically implemented system which serves as proof-of-concept. As an example, the user draws a class diagram from the UML. The system then derives the semantics of the diagram, and creates skeleton class files. The user can subsequently create an actual implementation with these skeletons.

Reaching this goal depends on two subsequent stages, applied after the user is finished drawing. The first is recognition, which means to identify the single shapes that make the complete diagram. The other step is analysis, which means to inspect each shape in the context of other shapes, thus being able to derive a syntactical structure first, and the semantics afterward. Recognition is subject to much current research in the field of sketching. Still, no satisfying solution could be found yet. State-of-the-art approaches mostly constrain the user and impose restrictions regarding how to draw. Thus, the task of recognition is simplified to a point where it becomes bearable, but the user is forced to concentrate on his drawing style. Analysis, on the other hand, is rarely discussed in publications on sketching. However, analysis should be an important aspect of approaches to

sketching, as the user is usually not interested in recognized shapes, but in the semantics of the diagram.

The approach presented in this thesis marks improvements, both for recognition and for analysis. The core idea of the recognition is to avoid a feature-based approach for high-level recognition, as features impose severe restrictions on which drawings can be recognized. Instead, a set of independent models is created, all of which contain information gained from low-level processing. Furthermore, multiple representations of the same stroke in different models are possible. This has a positive effect on recognition, because it removes the task of low-level processing to decide for a suitable representation without any context knowledge. High-level recognition itself is then based on composition of primitives to complete shapes. In general, the presented approach to recognition does not constrain the user in the ways shown by previous work in the field.

Analysis builds upon the DIAGEN framework, which allows for generation of WIMP-based diagram editors from specifications. The generated editors allow for checking syntax and semantics of the diagrams created by the user. The concept of DIAGEN is based on the formal approach of graph transformation, which results in a powerful and reliable diagram analysis.

In this thesis it is shown how this approach can be transferred to sketching. The most distinct result is that ambiguities can be reliably solved by extensive use of context information gained from syntax checking. Ambiguities naturally arise from hand-drawing, which is inevitably sloppy and imprecise. Furthermore, it has proven valuable to explicitly model ambiguities in the analysis process. Also, diagram language-specific output can be generated as a result from the analysis as motivated above with the example of class diagrams.

The prototypical implementation is applied to six different diagram languages, all of which exhibit different characteristics regarding visual appearance, syntax, and semantics. Among the six languages there are statecharts from the UML, and a GUI builder as a representative of DSLs. Many further diagram languages are conceivable as well. An empirical user study evaluates recognition rates and performance of the prototype. It proves that the system is both accurate and powerful.

The contribution of this thesis lies both in the recognizer and the analysis. The recognizer allows for multiple representations of the same stroke at the same time, and is capable of identifying shapes from a complete drawing without prior assignment of strokes to shapes. The analysis is based on a formal approach. Ambiguities are solved automatically based on the syntactic structure of the diagram language. Therefore, ambiguities are explicitly modeled for the analysis.

Contents

List of Figures

List of Algorithms

List of Acronyms

BLD	Boolean logic diagram
CNF	Chomsky normal form
CYK	Cocke-Younger-Kasami (parser)
DAG	directed acyclic graph
DSL	domain-specific language
EPR	embedding production rule
GUI	Graphical user interface
HCI	Human-computer interaction
HM	hypergraph model
HMM	Hidden Markov Model
LHS	left-hand side
MDA	Model Driven Architecture (http://www.omg.org/mda)
NAC	negative application condition
NPR	nonterminal production rule
NSD	Nassi-Shneiderman diagram
OMG	Object Management Group (http://www.omg.org/)
RHM	reduced hypergraph model
RHS	right-hand side
SPR	set production rule

TPR terminal production rule

UML Unified Modeling Language (http://www.omg.org/uml)

WIMP window, icon, menu, pointing device

Chapter 1

Introduction

Diagrams are widespread in computer sciences, as well as in other disciplines. Everybody uses diagrams: architects, engineers, sales people, and designers. This is no surprise, since a diagram naturally allows for representing information in a visual manner. This makes it easy to grasp its meaning. Perception of information which is clearly arranged graphically is mostly more convenient and quicker than that of a textual representation. Especially for very complex issues, using diagrams cannot be abstained from. Complex software could not be build without the use of diagrams, devices based on micro-electronics like mobile phones could not, cars could not, constructions like buildings or bridges could not.

Many popular and well-known types of diagrams are used in computer sciences today, although their use in other fields outnumber the usage in computer sciences by far. Common to all diagrams is that they have a certain meaning. In computer sciences this meaning is usually described (more) formally, as this discipline has the means necessary at hand. Similar to textual languages which can be described by string grammars, diagrams are related to diagram languages, which can be described formally as well. For example, in version 2 of the UML the OMG defined the meaning of respective diagrams more formally and clearly as before, which fosters using and understanding of UML diagrams.

Having a clear understanding of diagrams is essential to the processing of the contained information. Based on the UML, the OMG thus could establish MDA as a new paradigm to the production of software, which is – in its basic idea – completely based on models. These models are usually represented graphically as diagrams. This whole concept would fail if diagrams could not be formally described.

Of course, not every visual representation of information is a diagram. In this thesis, we understand a diagram always in the context of its diagram language. The language, as said before, describes a set of diagrams. Part of this description is syntax and semantics of the diagrams, no matter whether it is given formally or

not. If there is no language, we do not speak of diagrams.

Creating a diagram can serve two purposes. First, an already established circumstance should be illustrated. This is often the case, for example, when an author writes a book and uses diagrams to convey information or foster understanding, or when a software engineer documents the architecture or structure of software which is already built.

The other application of diagramming – to create and use diagrams – is to support the creative process which is inevitable when something new is to be created [63]. For example, designers of software usually start building the software by working with drafts first, and refining them until they meet certain criteria of quality or content. Only then the software is actually implemented.

For both purposes there exist countless tools, both commercially available and in academia. Examples of the former are *Borland Together*, or *IBM Rational Rose*. Examples from academia are DIAGEN [69], *DiaMeta* [71], *VLDesk* [26], *AToM*^{3} [34], *Fujaba* [37], *Tiger* [36], *Pounamu* [105] and *Marama* [45]. All of these tools allow for creating diagrams and processing them in some way. The processing is especially useful when the information described by the diagrams is to be used in subsequent steps of a complex process. For example, much source code can be generated from UML diagrams, which is, by the way, the basic concept of MDA.

Although commercial tools are usually much more mature, they usually have a similar user interface like the research prototypes. Using the mouse and an empty canvas, diagram components can be placed on the canvas by a matter of clicking buttons. The user is greatly supported in this process by features like cut-copy-paste, undo and redo, saving and loading, automatic layout of diagrams, and much more. However, such interfaces still remain artificial in some way, and are more dictated by technical aspects rather than by the users' needs.

It is well known that many designers prefer to use pen and paper in early stages of design, and not a computer. This has several reasons. First of all, it is easier. The paper does not need an explicit user interface, and it provides the user with much flexibility. This again fosters creativity, as no time and effort has to be spent in using pen and paper. Even more, when a computer is used, people tend to get distracted from their actual task, concentrating on minor aspects like a nice layout. Sketches made on paper look informal and unfinished, which clearly indicates that they are subject to change. A neat computer-generated layout has the opposite effect. People are more reluctant to change such diagrams [14, 63, 30, 79].

Designers usually begin with some informal sketches to talk about the design of a new car, for instance, and so do software engineers. Alone or in a group, the basic architecture is often developed in terms of informal sketches, which is simply more convenient. Of course, when using sketches in the first place this means that at some point they have to be transferred to a computer, often manually,

to gain further value [40]. Unfortunately, this process is both cumbersome and error-prone. Furthermore it is usually irreversible [42] as updates to the transferred diagrams can hardly be reflected in the sketches.

The research topic of *sketching* now deals with exactly this situation. Users are allowed to draw (or sketch) on a computer, and the machine interprets this input in some way. This interpretation is crucial and states the difference to bare drawing tools. The basic idea is to get the best from both worlds: a very simple and natural user interface as known from pen and paper [91, 38, 53, 4, 48, 30], but also capabilities to understand the sketches, plus comfortable editing commands as mentioned above (load, save, selection, versioning, etc.).

Because a mouse is no suitable replacement for a pen (nor is a touch pad like most notebooks are equipped with), special hardware is required for sketching. Examples are smart boards which can be used with the finger or any other suitable object, computers and notebooks with touch-sensitive displays where mostly a special stylus is required, and pen tablets which have no own display. According to this broad range of hardware, prices also span a wide range, starting from less than 100€ for simple tablets like the *Wacom Bamboo*[1], up to more than 10.000€ for smart boards like the *SMART Technologies 2000i*[2].

Such hardware is increasingly popular, and is more and more widespread [76, 81]. The underlying technology enables a completely different paradigm of interaction with computers, compared to classical user interfaces relying on a mouse, also known as WIMP (window, icon, menu, pointing device). Accordingly, using such hardware also requires dedicated applications which fully support and utilize their special capabilities [93, 80]. A touch-sensitive display, for example, is nothing more than a nice gadget if there is no software which benefits from the stylus. Understanding of hand-writing is such an application, where the user really benefits from the stylus. Sketching is another. It is reported that users prefer to use a whiteboard or a tablet PC rather than a traditional tool [99].

In the last two decades, much progress has been achieved in the field of sketching. Complete systems have been developed, and dedicated solutions to special issues have been proposed. However, the recognition of sketches is still not solved satisfactorily, and is constantly regarded as a difficult task, if not the hardest challenge in the field of sketching [11, 43, 91, 4, 98, 16, 79]. Recognition means to identify single shapes, or entities, from the sketch. Also, reasoning about sketches, i.e., further processing of the information conveyed in a sketch, is in its infancy, and is not considered in many publications. In the next section and in Chapter 3 a reasonable selection of this previous work is addressed and discussed in more detail.

[1] see http://www.wacom.com/

[2] see http://smarttech.com/

In this thesis we present a novel approach to the understanding of sketched diagrams, called DSKETCH (*diagram sketching*). New solutions to both the recognition and the reasoning are given, which depend on each other, but could also be used in combination with other approaches. For the recognition, we rely on a new concept that avoids by design common problems of other approaches. For the reasoning, we rely on an existing framework for diagram analysis, which we adopted to the special characteristics of sketching. The basic concept of DSKETCH is explained in Section 1.2. Before, Section 1.1 defines basic terms and aspects of sketching, and describes some of the current issues. DSKETCH is briefly summarized in Section 1.3, and its major scientific contributions are highlighted. The final Section 1.4 outlines this thesis by summarizing each upcoming chapter.

1.1 Key Aspects of Sketching

In this section the general terms and definitions are discussed which have evolved for the field of sketching. The design space for an approach to sketching is explained. As briefly mentioned before, *sketching* means the understanding of a hand drawing.

The central term in sketching is that of a *stroke*. A stroke begins with the stylus put on the surface of the screen, and ends with the lifting of the stylus. Just like a mouse, the computer tracks the movement of the stylus by capturing the events generated from its driver. Technically, a stroke is a finite list of tuples (x, y, t) (called *samples*) where x and y are a pair of coordinates reflecting the screen position where the stylus was, and t is the time elapsed since the beginning of the stroke, usually measured in milliseconds. Some approaches even consider the pressure exercised with the stylus on the screen [72] to improve recognition accuracy, but not every hardware is capable of measuring this value. Mostly, pressure is only used to adjust how strokes are displayed - the thicker the more pressure there was exercised [74]. The least common denominator are x, y and t. These values can always be captured, and are assumed by virtually any approach.

The main issue in a sketching system is *recognition*, which means to identify from the strokes drawn by the user the single shapes a sketch is composed of. For that purpose it must be specified in advance what shapes are to be recognized. A *shape* is a basic building block of a sketch that depends on the domain and cannot be broken into sub-parts. In our case, as we assume diagram languages, the set of relevant shapes is always known. For Petri nets, for example, there are only four kinds of shapes which have to be recognized from the drawing: places, tokens, transitions and arrows. Text can be used in order to augment the sketch, in the case of Petri nets by defining weights of arrows and capacities of tokens. Sometimes, recognition is broken down into two levels, *low-level recognition* or

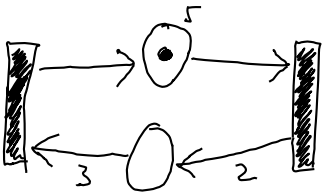


Figure 1.1: Example of a sketched Petri net.

low-level (pre)processing, and *high-level recognition*. The former means to extract intermediate information from the strokes drawn by the user, and the latter to use that intermediate information to identify shapes.

For various reasons recognition is very difficult. Consider the simple Petri net in Figure 1.1, for example. If 10 persons are asked to copy this diagram, the outcome will be 10 different results which are all sketches of the same Petri net. For example, one person draws precisely and neat, while another person draws imprecisely and sloppily. Or one person draws every shape in one stroke, while another uses two or more strokes. Another person heavily overtraces his own strokes. Even more so, if the same person is asked to produce 10 copies of the diagram, the outcome will still be 10 different sketches, because there is so much variation in hand-drawing. The bottom line is that the same diagram can be sketched in many different ways. This makes it very hard to develop approaches that reliably handle all of these different sketches properly.

Since the recognition of just a set of shapes is only the first step toward understanding of a sketch, a second stage should follow the recognition. This stage is the reasoning or *analysis*. Its exact task depends on the sketching approach and the domain. For example, analysis may relate the shapes to each other (e.g., which place contains which token), and process the resulting information.

Having strokes, the task of a sketching system is to assign a meaning to a given set of strokes by first recognizing shapes, and then reason about them. This is called *on-line* recognition, resp. an on-line approach. If no strokes are given, but a raster image, this is referred to as *off-line* recognition. Most current approaches rely on on-line recognition. Having information about individual strokes greatly aids in recognition, as it means more information compared to a raster image. If off-line information is available only, approaches like [81] can be used to extract information about strokes from the raster image.

The different approaches to recognition all rely on particular assumptions; some approaches assume more, others less. Obviously, depending on the assumptions, recognition can be simplified a lot. Approaches like [83] require each stroke to represent exactly one shape, for example. This way, two very difficult issues are avoided: clustering and segmentation. *Clustering* (also referred to as *grouping*)

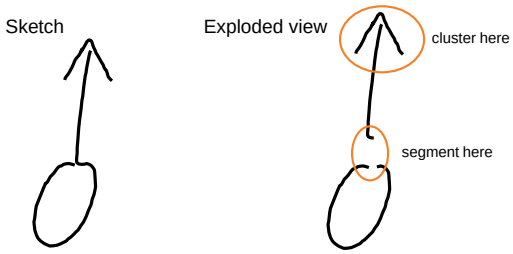


Figure 1.2: Example of two strokes which have to be segmented and clustered to yield shapes.

is the question which strokes have to be grouped in order to represent a shape, while *segmentation* (or *fragmentation*) concerns to split strokes if they contribute to more than one shape. The difficulty of both clustering and segmentation is that actually the recognized shapes are required to decide about them, but they are required to recognize the shapes; a typical chicken-and-egg problem. It is even possible that some strokes have to be clustered, while others have to be segmented in order to identify one shape. An example is given in Figure 1.2. The sketch comprises two strokes; circle and arrow shaft are drawn in the first stroke, while the arrow head is drawn in the second one. The exploded view shows how the first stroke must be segmented to separate the shaft from the circle, and how the shaft and the second stroke must be clustered to yield the arrow.

A *gesture* is understood as one stroke that follows a certain, predefined path. Gestures can be used to invoke commands, like erasing some part of a sketch, or selecting something. Although sometimes the case, gestures are not well-suited as replacement for shapes, as the look of a gesture often is quite artificial and does not look like the intended shape.

The publication *Specifying gestures by example* by Dean Rubine [83] approaches gestures by a feature-based recognition approach. It is of special importance, as it is often cited and has a great impact on the field. The basic idea is very straightforward and follows the classical methodology of pattern recognition. It is adopted by other approaches, which also use it for shape recognition. The central idea of feature-based recognition approaches is to train the system by drawing each shape that should be recognized one or more times. Rubine assumes exactly one stroke per shape. For each training stroke some features are computed and stored, e.g., the length of the stroke, the size of the bounding box, the average speed of the stylus, or the number of bends. If the same shape is trained more than once, the feature values can be averaged, for example. Rubine used 13 features; nowadays much more features are known in literature [87]. When the actual sketch is drawn, for each stroke the same features are computed as for the training strokes. Then, using a classification scheme, the system compares the feature val-

ues gained from training to those computed from the strokes in the drawing, and decides which gesture (or shape) is best represented by a stroke.

For example, in order to distinguish small circles from large circles, the stroke length can be considered as the only feature. For training, circles from both groups are drawn, and the respective stroke lengths are averaged. Then, for each stroke in the drawing, its length is computed, and it is classified to be a member of that group (small or large) where the average stroke length is closer to the one computed for the stroke in the drawing. While this approach is very simple, it is not sufficient in practice; more features should be used, as the stroke length alone is not sufficient to decide whether a stroke forms a circle at all, for example.

The advantage of feature-based approaches is that they can be easily implemented and work quite fast. The recognition rate, i.e., the number of strokes that are recognized correctly compared to the number of strokes that are drawn, depends very much on the features and on the classifier. Various further publications by other authors pick up this approach [79] and improve it in some ways, e.g., by computing other features [38, 76, 77]. It is also tried to get rid of the severe limitation that a shape has to be drawn in one stroke, which means that at least clustering is possible. However, many approaches are not very sophisticated in doing so, but rather rely on very simple rules to detect when the drawing of one shape has finished and the drawing of another begins. For example, a constant time threshold can be used, as in [6, 43, 14]. Whenever the time passed between the drawing of two consecutive strokes exceeds this threshold value, the strokes are considered to contribute to different shapes. Although simple to understand, [99] suggests that it is difficult setting the threshold value to a value that pleases different users. In [15] the user has to explicitly tell the system that a shape is completely drawn by pressing a button shown in the UI.

In general, for feature-based approaches clustering and segmentation are very hard problems. Even more, the classical approach by Rubine requires the user to draw the same stroke each time to be correctly recognized. For example, if the training strokes for a rectangle each begin in the upper left corner of the rectangle, and draw the shape clockwise, each later stroke that is supposed to be recognized as a rectangle must follow this rule exactly. This is a severe restriction in terms of how shapes are allowed to be drawn, and is unrealistic [64]. The same behavior can also be observed with Graffiti used to recognize hand-writing gestures for devices by Palm[3]. Work published which suggests that this limitation can be relaxed is usually based on more complex features like high curvature [76], or on continuous data like pen speed [40]. Overcoming these limitations is crucial, as [53, 74] argue that people indeed draw more than one object with one stroke, and that one object is not always drawn with a sequence of consecutive strokes.

[3] see http://www.palm.com/

The insight gained from feature-based approaches reveals two characteristics of sketching systems. First, they can be distinguished according to the number and kind of restrictions that are imposed on the user. Second, and closely related, recognizers are said to be either *single-stroke* or *multi-stroke* to indicate their capability to cluster strokes.

Inevitable for hand-drawing is a lack of precision. Many shapes are drawn somewhat messy, when the user is not very keen to produce an exact drawing. This imprecision naturally leads to *ambiguity*, i.e., the recognizer identifies different competing shapes which cannot exist in parallel. Feature-based approaches are usually more forgiving with sloppy sketches, which is a big advantage. However, even here ambiguities may arise. Mankoff et al. discuss how to overcome ambiguities, e.g., by providing the user with a list of possible alternatives and let him decide, or by having the user repeat the input [67]. However, an automatic decision would be better, so that the user does not have to be bothered by the shortcoming of the system to make the decision on its own. Automatic resolution of ambiguities usually involves examining the *context* of a shape, i.e., other shapes close to the ambiguous shape. Given that there are some basic rules how shapes can be connected to each other, some (or even all except one) ambiguous alternatives can be excluded. As an example, consider Petri nets again. If there is a shape that could either be a place or a transition, by examining the context it may turn out that this shape is connected to a transition by an arrow. Hence the ambiguous shape must be a place, because no two transitions must be connected by an arrow in a Petri net. As already mentioned, imprecision and sloppiness is also the reason why recognition is so difficult. If a hundred people are asked to draw a rectangle, it is highly unlikely that there are even two identical strokes (or sets of strokes, in the case of multi-stroke recognition). However, each of the hundred rectangles is expected to be correctly recognized, in spite of the large variance of input.

Another design alternative for sketching systems is whether to start recognition after every stroke (*eager recognition*), or to require the user to explicitly start recognition (*lazy recognition*). The former requires a very fast recognition, and is common for many feature-based approaches. The advantage is that the user may be given immediate feedback, which helps him to correct input that has not been recognized as intended. Also, editing of the sketch can become easier if shapes are already known [7]. On the other hand, it is reported that immediate feedback like beautification distracts the user from his actual task of drawing a sketch [3, 4, 39] (*beautification* describes the task of replacing the user-drawn shapes by neat computer-generated ones). Furthermore, as described above, the context of a shape may be required to solve ambiguities. Starting recognition after every stroke obviously means that no context is present for the shapes drawn first. The results of this recognition are not likely to be valid. Yet another issue, that of incremental recognition, is linked to eager recognition. Recognizing the whole sketch each

time from scratch degrades performance and hinders eager recognition. However, with incremental recognition available, eager recognition becomes possible.

Regarding user interfaces there is a clear distinction between *mode-less* and *mode-based* approaches. Other tasks than sketching itself, e.g., editing a diagram, can be performed at any time in a mode-less environment. For example, shapes can be erased at any time by scribbling zigzag lines over them. On the other hand, in mode-based approaches there is the need to explicitly change the current mode first, for example from *sketching* to *editing*. Only then a shape can be erased. When the user has finished erasing, he must switch back to the sketching mode prior to continue drawing. While this seems more complicated, it avoids any ambiguities involved in whether a stroke is supposed to contribute to the sketch itself, or if it represents an editing operation. Generally, modes are to be avoided [85], although they simplify recognition.

Regarding text, which is very important in many diagram languages, there is either the option to allow for writing directly on the canvas, which is easy and natural, or to require the user to indicate writing, which again is mode-based. Direct writing, however, involves the difficult task of distinguishing between text and graphics, which is still an open issue [75, 10, 79].

1.2 Concept of the Proposed Approach

The approach to sketching, DSKETCH, discussed in this thesis follows two main goals. All design decisions and key features originate from and can be motivated by these two goals. The first is to create a sketching system for the understanding of hand-drawn diagrams. The importance of diagrams and diagrammatic representation, especially in the context of domain-specific languages (DSLs), has already been highlighted above. Therefore we think that understanding hand-drawn diagrams is a valuable task that warrants research effort. The second major goal of DSKETCH is to allow for natural and unrestricted drawing. Sketching is, by itself, a natural input metaphor. This key attribute must not be violated or compromised by artificial restrictions; otherwise the whole idea of sketching is undermined. As mentioned before, the major issue of sketching is that of recognition. To tackle this challenge, all existing approaches make assumptions about certain aspects of the sketching process. If these assumptions go too far, or are unrealistic, they become restrictions or even limitations. The user interface no longer feels natural, and the user has to focus on his style of drawing, rather than on the drawing itself. This also prevents sketching systems from being used as mainstream technology [76]. As a consequence, there must be made a careful trade-off between the assumptions and recognition capabilities [4]. Other authors also admit the need for drawing in an unrestricted manner [91].

Based on these two major goals of DSKETCH, the remainder of this section first examines their implications on the overall design of the approach, and then outlines our solution as presented in this thesis.

There are two related issues which are neither solved nor discussed in this thesis. The first concerns the GUI of a sketching editor. In order to support the natural way of interaction introduced by sketching, the GUI should provide the user with respective functionality which seamlessly integrates with this way of interaction. For example, there should not be any gap for the user between sketching and invoking commands like load and save, or editing capabilities. In general, designing a good UI is difficult [12], and is research topic of the HCI community. The design and evaluation of a GUI is not discussed in this thesis. The second issue is the separation of text from graphics. In the ideal case, the user can write text right on his sketch. Then the system automatically distinguishes strokes representing text from strokes representing graphics. However, this task is very difficult, and not solved satisfactorily yet, as described above. We assume that text and graphics are already separated by some way, e.g., by a mode-based GUI.

Implications by the major goals

This subsection first discusses the implications of the major goal of understanding hand-draw diagrams, and then does so for the other goal of natural, unrestricted drawing.

The first implication of having hand-drawn diagrams is to use on-line recognition. The goal is to replace traditional widget-based editors and their point-and-click input metaphor by a sketching approach. Accordingly, the user's strokes can be directly captured, and information on individual strokes and timing is present.

The second implication is to design an approach that is fully generic. The advent of DSLs is a central motivation, so the approach must be capable of understanding diagrams from different domains. Furthermore, for the field of classical diagram languages like the UML, or other examples mentioned above, only a generic approach can be used in order to understand the very different diagrams these languages entail.

The third implication also lies in the nature of diagramming. Nobody draws diagrams for its own sake; there is always the desire to convey information with a diagram. The power of diagrammatic representation stems from the information contained in diagrams [42]. Even more, when a computer is used as drawing tool, there is usually the intention to make the computer understand the diagram, and use the contained information for some task. For example, when drawing a Petri net on a computer, the user may want to execute the net to see if it matches his understanding of the problem he modeled. Accordingly, using DSKETCH it must be possible to compute a domain-dependent output as the result of processing a

hand-drawn diagram. The need for a flexible output is also mentioned in [79].

These three implications are due to the goal of understanding sketched diagrams. The next four implications result from the other major goal, natural and unrestricted drawing.

The fourth implication is to support multi-stroke shapes. Depending on the diagram language, shapes may have a complex visual appearance, where it is bothersome to draw such shapes in one stroke. Besides, even if the shapes are simple, it is more convenient to draw them in as many strokes as one desires.

The fifth implication regards the technique used for recognition. The previous section has discussed the pros and cons of feature-based recognition. It can be easily implemented, but usually makes certain assumptions on the user's style of drawing. Accordingly, we believe that such approaches should be abstained from, at least in general. For special cases, however, feature-based recognition may be useful and should be considered.

A central issue of feature-based recognition is that clustering, and even more so segmentation, are very hard challenges. However, both must be possible for a sketching approach in order to not restrict the user while drawing. Therefore, the sixth implication is to allow for automatic clustering and segmentation of the user's strokes.

The seventh and final implication is to solve ambiguities automatically. Ambiguities cannot be avoided, so they must be dealt with. If this involves the user, e.g., by asking for the correct interpretation of an ambiguous shape, naturalness is lost, and the interface becomes more cumbersome and artificial.

In order to conclude this subsection, the following enumeration lists all seven design implications. Keep in mind that the first three are due to the goal of understanding hand-drawn diagrams, while the last four are due to the goal of unrestricted sketching.

(i) on-line recognition

(ii) a generic approach

(iii) domain-specific results

(iv) multi-stroke recognition

(v) no dependency on a feature-based approach in general

(vi) automatic clustering and segmentation

(vii) automatic resolution of ambiguities

The next subsection discusses the design of DSKETCH in order to meet these implications.

Deriving design from the implications

This subsection gives a conceptual overview of DSKETCH. The approach itself is designed in a way that all implications outlined above are considered.

Implication (ii) requires designing an approach that is generic, i.e., not tailored to a specific diagram language. Instead, it must be possible to customize the approach in order to adapt it to a diagram language. This customization, which has to be done by a language designer, must be *specified* in some way. So the key to a generic approach is to have specifications that describe all necessary customization. Of course, the approach must also be equipped with an interface where the specification is entered. More details on specifications are discussed in a subsection below.

Implications (iv) through (vi) demand multi-stroke recognition, no dependency on features, and support for automatic clustering and segmentation. All of these issues concern the recognizer. On the other hand, ambiguities (vii) cannot be solved until all shapes are recognized, and a processing result (iii) cannot be computed before all shapes are known. Both tasks can be summarized by the term *analysis*. Consequently, there must be a second stage following the recognition which performs this analysis.

Figure 1.3 shows the bisection into recognition stage and analysis stage, and the specification. Processing steps are shown as rectangular boxes, data structures are shown as rounded boxes. Solid arrows denote flow of information; the dashed arrow denotes the influence of the specification. From the user's perspective, there is a sketching editor that provides a GUI where he can draw the sketch, while the output of the editor is the domain-dependent result (iii). Internally, the GUI captures the strokes drawn by the user, as required by implication (i), and captures the text written on the canvas, and feeds the recognition with the strokes and text. Then, the recognition stage identifies all shapes represented by the strokes and text, and the analysis stage computes the final result based on the shapes. The full process is customized by the specification.

Recognition and analysis

For the recognition we propose an architecture that does not solely rely on features. Instead, a different architecture is used, as shown in Figure 1.4. Whenever a stroke is drawn on the canvas, it is unknown what this stroke is supposed to represent. Also, segmentation and clustering are unknown. To account for this uncertainty, all strokes (and text) are first fed into *transformers*, which perform low-level preprocessing. The reason for having more than one transformer will be explained below.

The results of the transformers are stored in *models*. Each transformer is as-

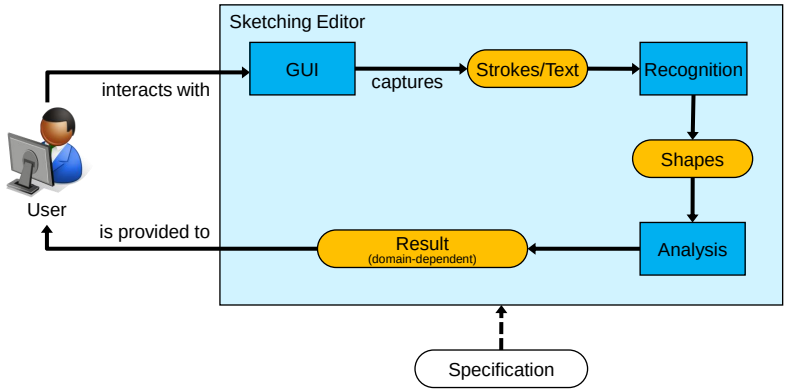


Figure 1.3: Conceptual overview of the full approach.

signed to one model. The models contain information abstracted from the strokes. All transformers are independent of each other, which allows for different interpretations of the same stroke in parallel. The *assembler*, the heart of the recognition stage, then derives a set of preliminary shapes from the models. The key idea to reach implications (iv) and (vi) is to abstract all strokes to a point where the contents of the models no longer represent individual strokes. Then, the actual recognition performed by the assembler is inherently multi-stroke, and performs clustering and segmentation on the fly. Furthermore, the transformers are implemented in a way that they do not rely on feature-based recognition in general (v). Finally, the set of preliminary shapes yielded by the assembler is postprocessed. The result of the postprocessing is again a set of (altered) shapes. The main task of postprocessing is to reduce the number of shapes that are passed to the analysis stage, which saves processing time of this stage.

As mentioned in Section 1.1, ambiguities naturally arise in sketching because of sloppy user input. Either the user can select the appropriate interpretation in case of ambiguity, or the system can make this decision on its own. According to implication (vii) we apply the second alternative, as it results in less distraction of the user, and enables a smoother flow of interaction which feels more natural. The idea of the analysis stage is to use the context of an ambiguous shape in order to determine the correct interpretation. This is the same behavior as shown by humans [38]. Context is given by other shapes spatially close to the ambiguous shape. In our case, as we deal with the understanding of diagrams, valid contexts of a shape are defined by syntactical rules describing the diagram language.

As there are powerful frameworks to reason about syntax and semantics of diagrams, we decided to use such a framework as basis for our analysis stage, as opposed to creating a new solution from scratch. For this purpose we have decided for DIAGEN [69, 68], which allows for generation of diagram editors

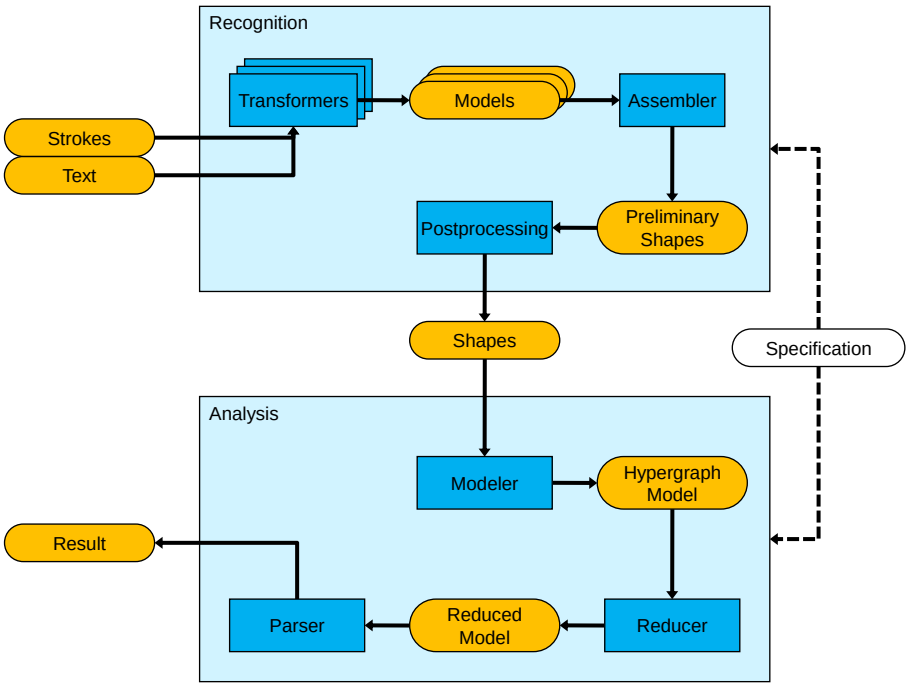


Figure 1.4: Conceptual overview of the recognition stage and analysis stage. All individual steps are shown. This figure refines Figure 1.3, user and GUI are not shown. The specification affects each of the six steps.

based on specifications. The generated editors internally use hypergraph models and allow for very efficient graph parsing of the diagrams to generate a derivation structure that reflects the syntactical structure of the diagram. Furthermore, and indispensable to DSKETCH, the diagrams can be created by *free-hand editing*, which means that all diagram components can be freely arranged on the canvas. The alternative to free-hand editing is *structured editing*, which means to edit diagrams by predefined editing operations. However, this alternative does not correspond to sketching.

The analysis stage is composed of three steps (cf. Figure 1.4), all of which are based on DIAGEN, but adapted to the needs implied by our application to sketching. Each of the three steps is guided by the specification. First, the *modeler* creates a graph structure that represents all shapes and spatial relations between these shapes. Examples for spatial relations can be seen in Figure 1.1, e.g., the token that is *contained in* the place. Due to its nature, this graph model is called *hypergraph model* (HM). In the second step of the analysis stage the *reducer* takes the HM and creates another graph model called *reduced hypergraph model* (RHM). The idea is to reduce the size of the HM, and to filter out invalid patterns, e.g., an arrow in a Petri net that connects two places. Finally, in the third step the *parser*

performs bottom-up hypergraph parsing, using the edges in the RHM as terminal symbols. The task of the parser is to identify a syntactically and semantically correct subset of all shapes identified by the recognizer. The derivation structures obtained by the parsing process are used to compute the final result of diagram processing, described by implication (iii). Hypergraphs are used by DIAGEN as data structure, as they are very suitable for the purpose and allow for an easy, intuitive modeling of diagrams. Accordingly, both the reducer and the parser are controlled by rules, and work similar to existing graph transformation systems.

The existence of an analysis stage suggests abstaining from eager recognition; instead, lazy recognition should be preferred. The analysis heavily exploits context information, thus it is not meaningful to start processing a sketch before drawing has finished. Only the user knows when this is the case, so he is the one to decide about when to process his sketch. A similar approach is taken by [42].

Specifying a diagram language

As mentioned above, DSKETCH is not limited to a specific diagram language or domain. Instead, it is generic and can be customized using a specification of a diagram language to understand exactly this language. The specification is written by a language designer. It contains all the information required to understand a sketch. This includes the visual appearance of shapes, how they can be related to each other, and further information describing syntax and semantics, which is required by DIAGEN. Syntax is described by reduction rules for the reducer, and production rules for the parser, while semantics are specified by rules for attribute evaluation applied to the derivation structure obtained by the parser.

When processing a sketch, shapes are recognized first, and then analyzed. Each of these two stages is comprised of three steps. Thereby, the specification only describes *how* the individual processing steps are applied, but not *if* they are applied. Fact is, each processing step is always applied, and never left out.

The visual appearance of shapes is a central issue for DSKETCH. Since the whole concept is generic, there is also a need for a generic framework that allows for specifying shapes from different domains. The basic idea, similar to other approaches, is to construct shapes from *primitives*. Primitives are the basic building blocks for shapes. They do not depend on any diagram language or domain. In order to identify a reasonable set of primitives we have examined various diagram languages, e.g., the UML. We found out that a broad range of shapes can be broken down to a small set of primitives. These must be kept very simple in order to be not too specific. In total we identified four different primitives suiting our purpose. These are straight lines, arcs, links and text. Examples of shapes and primitives are shown in Figure 1.5.

Straight lines are the most often used primitive. A rectangle, for example,

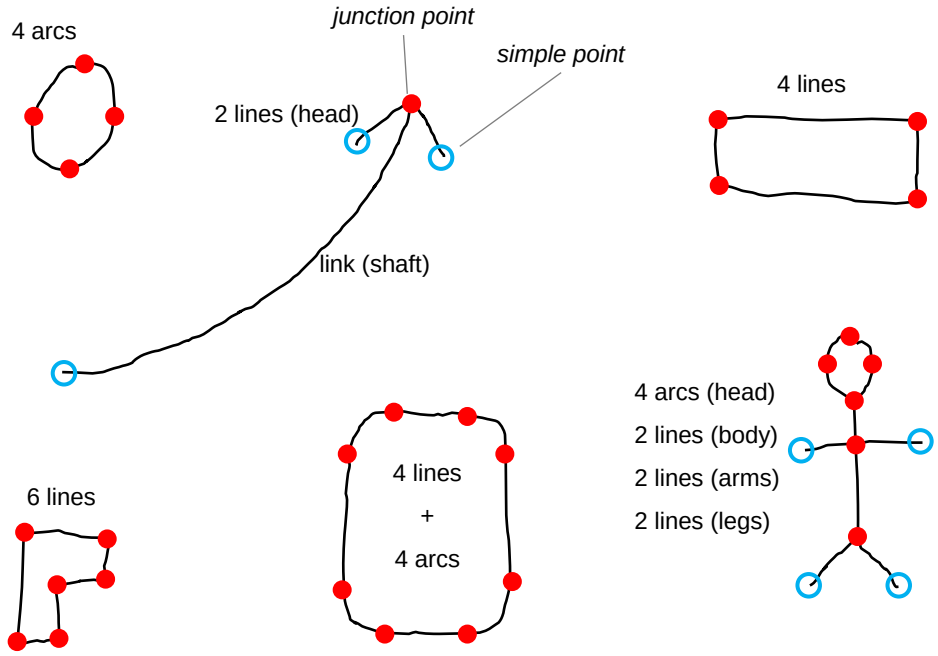


Figure 1.5: Examples of shapes, and how they are composed of primitives. Junction points are shown as filled circles, simple points are shown as unfilled circles.

consists of four straight lines, which are connected at the corners. Straight lines (or simply lines) can be distinguished by their orientation, i.e., horizontal, vertical, diagonal ascending, or diagonal descending. For example, an axially parallel rectangle consists of two vertical and two horizontal lines. The legs of an UML actor symbol are given by two diagonal lines. Sometimes it is even necessary to allow an arbitrary orientation for a line, e.g., when a rectangle is specified that should not be restricted to be axially parallel.

By arcs we assume quarters of circles or ellipses which are located in one quadrant of the coordinate space only, for example, from 12 o'clock to 3 o'clock. Accordingly, there are four different arcs we distinguish, as there are four quadrants. A circle, for example, is made of four arcs. A rectangle with rounded corners consists of four straight lines for the four sides, and four arcs for the four corners.

Links connect two points by an arbitrarily shaped line, which may be bent or exhibit curves, and which may also be straight. For example, an arrow head may consist of two straight lines, whereas the shaft may be a link. Similarly, all kinds of associations between classes in UML class diagrams can be made of links to allow for more flexibility in drawing. While the line primitive describes only straight lines, and the arc primitives describes only arcs located in one quadrant,

the link primitive does not describe such restrictions, but is more general.

These three primitives have in common that they each connect two points. This is an important observation for the specification of shapes. For the rectangle, each of the four corners is a point that is connected to two other points by two lines. The tip of the arrow is a point where both the lines for the arrow head begin, and where the shaft begins. We call such points that are connected to other points by at least two primitives *junction points* to indicate that two (or more) primitives are joint at these points. Figure 1.5 shows examples of shapes and how they are composed of primitives. Junction points are shown as filled circles. Points which are no junction points are shown as unfilled circles. We call these points *simple points*.

Of course, primitives alone are not sufficient to specify shapes. Using *constraints* we can further specify primitives. For example, the arrow head has to be constrained in such a manner that not every two lines and one link connected at one junction point (the arrow tip) form an arrow, but only if the angles enclosed by the lines of the head and the shaft are not too large, and if the lines of the head have the same length, and if the shaft is considerably longer than the lines of the head.

The fourth primitive, text, represents text strings that have been written on the canvas. Internally, the text primitive is characterized by the string itself and the bounding box describing where the text is located on the canvas. To specify how the text primitive is related to other primitives, the bounding box of the text is related to junction points and simple points of these other primitives.

Since text conveys part of the meaning of a sketch, it is often restricted in its content. For example, the capacity of a place in a Petri net is described by a number, and not by letters (cf. Figure 1.1). Therefore, for an implementation it may be useful to provide means to describe the contents of a text, e.g., by a dictionary, a string grammar, or a regular expression.

Due to these four primitives we propose to use different transformers in the recognition stage. This way, each transformer can process strokes from a different point of view. One transformer only looks for straight lines, another only for arcs, and yet another for links. This allows for separation of concerns; however, there is no need for a one-to-one correspondent between primitives and models. Finally, further transformers can be integrated, e.g., to improve low-level recognition by applying different algorithms.

A different point of view

The concept of DSKETCH can be illustrated from a different point of view than the one taken in Figure 1.3. Instead of focusing on the concept of the approach, the technique to include the specification can be highlighted. It works in the following

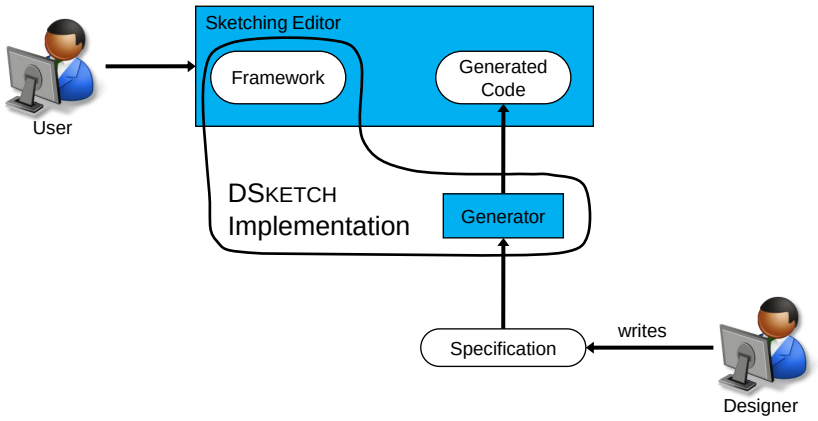


Figure 1.6: Influence of the specification on the sketching editor.

way. From the specification delivered by a language designer, source code is generated. This source code, together with the framework described in this thesis, constitutes the sketching editor, which the user operates to sketch diagrams. This concept is shown in Figure 1.6. We have created a prototypical implementation of DSKETCH which is used to validate and evaluate the presented ideas. Both the framework and the generator are part of this implementation.

1.3 Main Scientific Contributions

In this thesis an approach to sketching is presented. Based on the specification of a diagram language, a sketching editor is generated. The editor consists of a GUI and processing capabilities. A sketch is processed in two steps. First, there is recognition, and then follows analysis. Analysis resolves the inevitable ambiguity of recognition outcome. The full approach is designed along two major goals: understanding of diagrams and preservation of the natural input metaphor of sketching.

The goal of this section is to outline the major scientific contributions of the approach presented in this thesis. Due to the large body of related work that exists in the field of sketching, there are many approaches capable of recognition. Accordingly, the contribution of a new approach is characterized by its unique features, which make the difference to previous work. Related work itself will be discussed in Chapter 3. A comparison to related approaches is drawn in Section 3.6, which is necessary in order to highlight the importance of our contribution.

The two major design goals and their seven implications, as identified in the previous section, represent a unique combination of features which cannot be found at any other approach. The seven implications are on-line recognition, a

generic approach, domain-specific results, multi-stroke recognition, no dependency on a feature-based recognition in general, automatic clustering and segmentation, and automatic resolution of ambiguities.

The overall contribution of this thesis is manifold. All of the following benefits have been achieved with the proposed approach. They are explained in detail throughout the thesis:

- The recognition process uses a new, powerful architecture. It relies on independent transformers and models, which allow for parallel interpretation of the same stroke. An automatically computed *search plan* determines the search order in which individual shapes are identified and assembled.
- The proposed recognizer is modular. Its design allows for easy integration of new primitives and constraints.
- There is a clear distinction between low-level recognition and high-level recognition. Low-level recognizers from other research groups can be integrated.
- There is an analysis stage, which uses rules given by the specification to automatically resolve ambiguities in a reliable and robust way. For this purpose, ambiguities are explicitly modeled in the analysis stage.
- The analysis stage is based on graph transformation, which has proven to be very useful for this purpose. There is no other generic approach we are aware of that does so.
- Both the recognition stage and the analysis stage comprise efficient algorithms, leading to a good performance of the full approach.
- The actual recognition rates of the approach are good, even for sketches from different domains.

1.4 Outline

The structure of this thesis is imposed by the two stages of diagram processing we have established, as shown in Figure 1.4. The six steps involved are discussed in Chapters 4 through 9.

In Chapter 2 some examples of diagram languages are given: Petri nets, Nassi-Shneiderman diagrams, a GUI builder, Boolean logic diagrams, Tic-tac-toe, and statecharts. All of these diagram languages have been specified and tested with DSKETCH. Furthermore, the chapter characterizes diagram languages DSKETCH can be applied to.

An in-depth discussion of related work is given in Chapter 3. A selection of closely related approaches is illustrated in more detail: the work of Hammond et al. (LADDER), Costagliola et al. (Sketch Grammars), and Plimmer et al. (InkKit) is discussed. Further approaches are addressed as well. A comparison of the approach presented in this thesis and the three explicitly mentioned related approaches is given.

The three steps involved in the recognition stage are illustrated in Chapters 4 through 6. Chapter 4 discusses the low-level processing which is performed. Here, strokes and text are processed by transformers, resulting in independent models. Chapter 5 explains the actual search process by the assembler, used in order to identify shapes. The search plan, which determines the search order, is explained in detail. This step is followed by postprocessing described in Chapter 6, where the results from the previous step are filtered and some extra information is added regarding ambiguities.

The analysis stage is also composed of three steps, discussed in Chapters 7 through 9. Chapter 7 deals with the creation of the hypergraph model from the shapes identified by the recognizer. In this step, spatial relations between shapes are established. Processing of the hypergraph model by the reducer is illustrated in Chapter 8. Its purpose is to reduce the number of edges and nodes in the graph model, and to discard invalid patterns. Finally, Chapter 9 explains the parser, which verifies syntax and establishes semantics based on the derivation structure, and generates the final result of diagram processing.

Chapter 10 discusses lessons learned from several case studies conducted with DSKETCH, and evaluates both the performance (processing time) and the recognition rate by means of a user study. The thesis is concluded with a summary in Chapter 11.

Appendix A gives the complete specifications of the example diagram languages from Chapter 2. Appendix B shows a figure containing all processing steps and data structures of the complete sketch understanding process. Appendix C illustrates a complete example of the processing of a diagram, using real-life data taken from the implementation.

Chapter 2

Diagram Languages

This chapter discusses some examples of diagram languages, all of which can be specified in and processed by DSKETCH. This means that the visual appearance of the shapes, the syntax, and the semantics of each of these languages can be described in DSKETCH, with suitable specifications. These diagram languages were selected due to their characteristic nature; they represent a wide range of diagram languages.

A classification of diagram languages is given in [28]. The main distinction is between (i) *connection-based* classes and (ii) *geometric-based* classes, each describing a hierarchy of single classes (cf. Figure 2.1). Furthermore, there exist hybrid languages. (i) is characterized by graphical shapes which are inter-connected in some way, e.g., by lines, or by arrows. Its two sub-classes are *Graph* and *Plex*, which both represent graph-like languages, but the latter is restricted to such languages where each shape is connected to a fixed number of other shapes. An example of class Graph are Petri nets (Section 2.1). An example of class Plex are Boolean logic diagrams, which are discussed in Section 2.5. Furthermore, this language serves as a good example, because the meaning of an operator shape is determined by an attribute (in this case, text written inside the shape). Among the presented selection of diagram languages, this applies to Boolean logic diagrams only.

For geometric-based classes (ii) the spatial arrangement of the shapes of the language conveys necessary information. Accordingly, relations between shapes can refer to adjacency, inclusion, or intersection, for example. The most general class within (ii) is called *Box*. Shapes of languages in this class are characterized by their bounding box, which does not need to have a fixed size for each shape. Similar is class *Iconic*, but it requires fixed sizes for the bounding boxes of shapes. Finally, there is class *String*, which contains all textual languages. These are not subject to our work. An example of class Box are Nassi-Shneiderman diagrams (NSDs), which are described in Section 2.2. An example of class Iconic

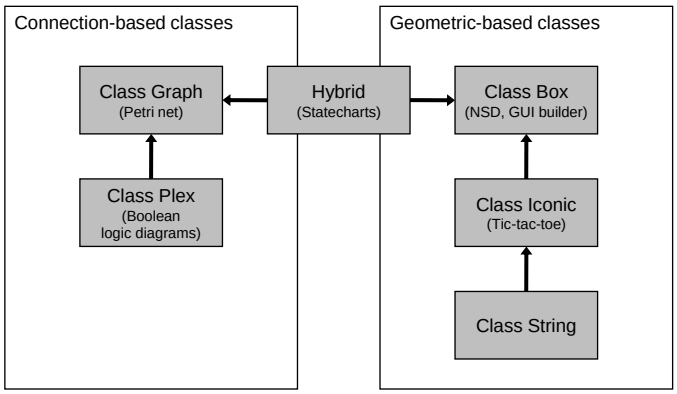


Figure 2.1: Overview of classes of diagram languages [28]. The arrows denote subset relations.

is the game Tic-tac-toe (Section 2.6). Although not a classical diagram language, and without technical application, it exhibits all characteristics in terms of syntax and semantics, and indicates that DSKETCH can be applied to a wide range of languages.

A representative for hybrid diagrams are statecharts. States are connected by arrows (class Graph), and hierarchical states are expressed by inclusion (class Box). Statecharts are illustrated in Section 2.4. Finally, we have included a simple GUI builder as an example, because many other approaches to sketching use this kind of diagram language as well, for example, *SILK* [61, 62, 63], *DENIM* [65], or *JavaSketchIt* [14, 59]. The GUI builder is another example for a diagram language in class Box.

The final Section 2.7 of this chapter discusses the application range of the proposed approach, and characterizes the diagram languages it can be applied to.

2.1 Petri Nets

Petri nets are used for modeling distributed systems. There are four different shapes: places, transitions, arrows and tokens. Arrows connect either a place to a transition, or vice versa, but neither two transitions, nor two places. Places are circles and may contain one or more tokens. Tokens are also circles, so the context of a circle (i.e., if the circle is inside another circle or contains another circle) must be used to distinguish places from tokens. Transitions are axially parallel rectangles, which are usually either small in width or in height. We neglect this circumstance and regard any rectangle as transition, which eases drawing. Furthermore, both transitions and tokens are usually completely filled. We neglect this, too.

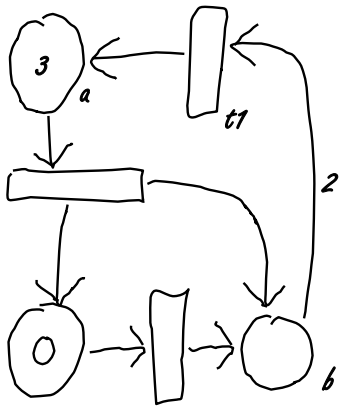


Figure 2.2: Example of a Petri net.

Text can also be used on Petri nets, but is optional. A number written inside a place denotes the capacity of the place, i.e., the number of tokens that may be contained. Arbitrary identifiers written near the border of places or transitions represent labels, which can be used to distinguish places and transitions textually. A number written close to the shaft of an arrow leading to a transition denotes the cost of the transition, i.e., the number of tokens required inside the place to fire the transition. A number written close to the shaft of an arrow leading to a place denotes the number of tokens that is added to the place if the transition actually fires.

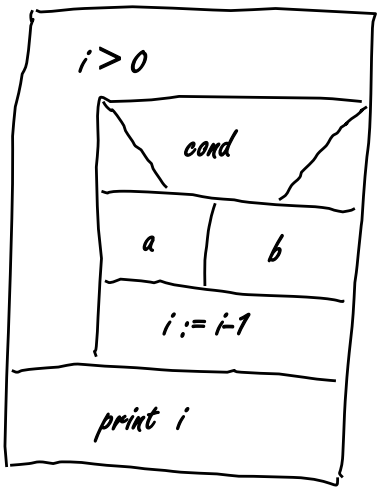
A transition may fire if each input place of the transition contains as many tokens as the costs of the arrows to the transition indicate and if the capacity of each output place is not exceeded by adding the number of tokens indicated by the number at the arrows from the transition. If several transitions can fire, it is non-deterministic which one goes first.

An example of a hand-drawn Petri net is shown in Figure 2.2. Two of the places are labeled (a and b), and place a has a capacity of 3. The arrow between place b and transition t1 has a cost of 2. Cost of an arrow is assumed 1 if not specified. Capacity of a place is assumed infinite if not specified.

2.2 Nassi-Shneiderman Diagrams

A Nassi-Shneiderman diagram (NSD) is a graphical representation of a structured program [73]. Its syntax allows for expressing statements, loops and conditions. The outline of each correct NSD is always a rectangle parallel to the axes. Note that we cover only a subset of the original language here.

An atomic statement is represented by a rectangle which contains some text



(a)

```
WHILE i > 0 DO
   IF cond THEN
      a
   ELSE
      b
   FI
   i := i - 1
OD
print i
```

(b)

Figure 2.3: (a) example of a Nassi-Shneiderman diagram. (b) corresponding pseudo code.

specifying the actual statement. Two statements which are to be executed one after the other are represented as two rectangles with the same width, one above the other, but not overlapping, such that their corners touch. Conditions evaluate a boolean expression, having two branches for the true and false case. The graphical representation is again a rectangle, this time with two diagonal lines inside the rectangle. Although possible with DSKETCH, we neglect the letters y (or t) and n (or f) which are usually written inside the triangles formed by the diagonal lines. This simplifies drawing without loss of any information, as the left branch is always considered to be the case where the condition evaluates to true, while the right branch is the case where the condition evaluates to false. Finally, loops are represented by shapes formed like the rotated letter L. They also contain text specifying the condition to be evaluated. Figure 2.3(a) shows a simple NSD consisting of four statements, one condition, and one loop. A string representation of this structured program is given in Figure 2.3(b) (keywords are written in uppercase letters).

There is a second type of loop, which differs from the shown while-loop in that it evaluates its condition only *after* the first run of the loop, so it is like the repeat-until-loop known from the programming language Pascal, for example. Its graphical representation in NSD is similar to the while-loop, but the horizontal bar (containing the condition) is below the enclosed block, instead of being placed above it.

Although cumbersome for larger programs, the benefit of NSD is its clear

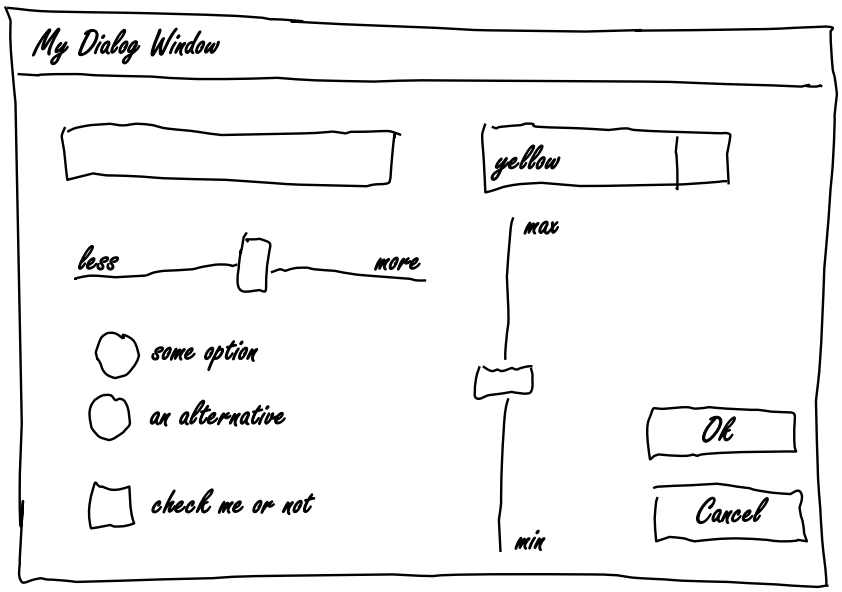


Figure 2.4: Example of a hand-drawn dialog box.

graphical representation, which allows for illustrating algorithms in a nice, visual manner. Therefore, NSDs are still widely known.

2.3 GUI Builder

A well-established, often used field of application for sketching tools are GUI editors or builders. The idea is to allow designers for drawing user interfaces, from which the actual GUI can be generated. While this application does not seem like a classical diagram language, a simple GUI builder can be built with DSKETCH as well.

The goal is to draw dialog boxes with basic, common controls. We decided for buttons, combo boxes, radio buttons, check boxes, horizontal and vertical sliders, and text fields. Buttons are drawn as rectangles with a label written inside. A plain rectangle without a label represents a text field. Radio buttons are represented by circles with text to their right; check boxes are represented by small squares with text to their right. Sliders are represented by small rectangles, placed either on a horizontal line, or on a vertical line. Either way, the ends of the lines represent the minimum and maximum values for the sliders. They may be labeled optionally. A combo box is represented by a rectangle with a vertical line and some text inside; the window is represented by a rectangle with a horizontal line below the window title. Figure 2.4 shows an example window containing all possible controls.

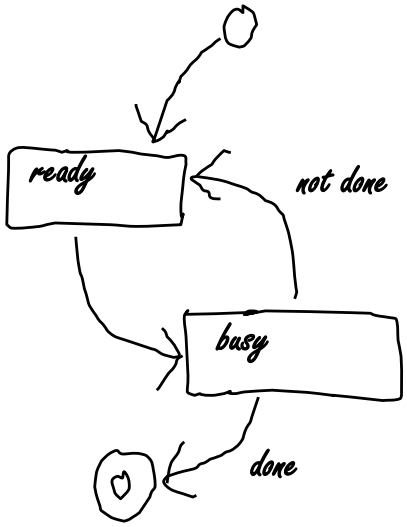


Figure 2.5: Example of a statechart.

As a result of processing a hand-drawn dialog box, different possibilities are conceivable. For example, the actual window can be generated, or source code to generate that window. Additionally, the window can be beautified, i.e., controls can be aligned, and fitted in size and position.

2.4 Statecharts

Among the examples in this chapter, statecharts are the only diagram language from the UML family. They represent the internal state of entities, and are typically used in conjunction with UML class diagrams. Similar to finite state automata, transitions between the states can be expressed, along with aspects like conditions for transitions, or actions which happen when a transition fires.

We cover a subset of the UML statecharts. States are drawn as rectangles. The name of a state is written inside it, just below the top line. Transitions are drawn as open-headed arrows. At the shaft, textual labels can be attached, containing information about signals, conditions and actions. The very first state of a statechart (the initial state) is drawn as a circle. It has to be unique for a statechart and has to have exactly one outgoing transition, unlike all regular states. Final states are drawn as circles containing another circle. Initial states must not be the sink of any transition, while final states must not be the source of any transition. An example is shown in Figure 2.5. It is possible (although not shown in the figure) to nest a complete statechart in another state. This way, the internal states of this state can be modeled, too.

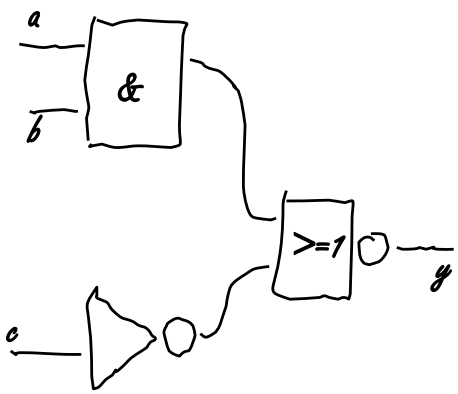


Figure 2.6: Example of a Boolean logic diagram. The diagram represents the term $y = \neg((a \wedge b) \vee \neg c)$.

2.5 Boolean Logic Diagrams

Boolean logic diagrams (BLD) allow for expressing boolean terms consisting of conjunctions, disjunctions and negations. An example is shown in Figure 2.6. Operators are drawn as rectangles. Inside the rectangle either & is written for a conjunction, >=1 for a disjunction, or 1 for a negation. Alternatively, a negation can be drawn as a triangle. For each operator, a small circle can be attached to the right. It is called a *bubble* and negates the output of the operator. For negation, the bubble is mandatory. In general, input to operators is indicated by lines ending on the left side of the operator, while lines indicating output begin on the right side, or on the right side of the bubble, if there is one. Input to an operator that is no output of another one represents and input variable and is labeled with a respective variable name. Accordingly, output of operators that is not input to another operator represents the result of the diagram and is labeled with the result variable name. Like NSDs for structured programs, BLDs represent their content in a visual appealing manner which can be easily understood, even at a first glance.

What is very special about BLDs are the labels written inside the operators (rectangles). On the one hand, the labels describe the actual operator, which means that text may also convey crucial information. On the other hand, the arity of the operator changes depending on the text. Negations are unary, while conjunctions and disjunctions are binary. This means that the information given by text can be used to disambiguate, too.

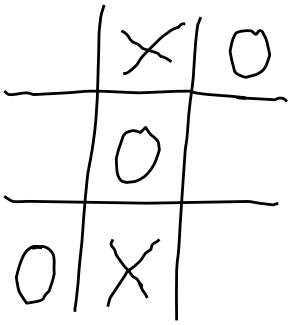


Figure 2.7: Example of a Tic-tac-toe game.

2.6 Tic-tac-toe

Tic-tac-toe is no typical technical diagram language, but can be specified using DSKETCH. The rules of the game are very simple. Two players play against each other on a 3x3 grid. In turn, they put marks on the fields of the grid; each field may contain only one mark. One player usually draws crosses, the other noughts. The first player with three of his own marks in a row (vertical, horizontal, or diagonal) wins the game. E.g., in Figure 2.7 the player drawing noughts has won. If all nine fields are marked, and there are no three equal markings in a row, the game ends in a draw. For this very simple game, it can be shown that the game always ends in a draw if both players play optimal, i.e., do not make errors which allow their opponent to win.

There are only three shapes to be recognized, the grid, the cross, and the nought. In general, the result of processing such a drawing could be an assessment of the game situation, e.g., whether one player has won, whether the game situation is legal, or whose turn it is. Also, game situations which are illegal, e.g., three noughts and only one cross, can be seen as semantic errors.

2.7 Application Range of the Proposed Approach

As discussed in Section 1.2, DSKETCH is designed for the understanding of hand-drawn diagrams. In this respect, an important question is the class or characteristics of diagram languages that the approach can be applied to. This chapter has shown six different languages which cover all classes of visual languages shown in Figure 2.1. In this respect, our approach supports a broad range of diagram languages.

The field of application is limited by two factors. The first is the expressive power of the production rules supported by the underlying parser. In general,

context-free languages can be described, and there is an extension which also allows for describing certain aspects which are not context-free. For example, arrows in graph-based languages like Petri nets, statecharts, or automata can be embedded this way. Details will be given in Chapter 9. Note that languages whose diagrams have a tree-like structure, e.g., BLD, can be specified with context-free rules only.

The second limiting factor is the expressive power of the specification of shapes. The graphical primitives like lines or arcs are predefined, but further primitives can be integrated in the given concept, if necessary. However, the specification assumes that shapes exhibit a fixed visual appearance that is free of structural variability. For example, a rectangle always consists of four straight lines, connected at their end points. While a rectangle can be drawn freely, for example, very thin or more like a square, and in one stroke or in several, the rectangle always consists of these four straight lines – the structure is fixed.

To conclude this section, DSKETCH can be applied to diagram languages from all classes illustrated in the introduction to this chapter, given that their syntax can be described context-free (or using the mentioned extension), and given that the visual structure of the shapes is fixed.

Chapter 3

Related Work

Approaches to sketching can be grouped according to their overall objective. There are those approaches which aim at the understanding of technical drawings, like we do, and there are other approaches, e.g., for modeling, for animation, or for design. The first group is strongly related to our work, and related approaches are discussed in this chapter. Approaches from the second group have a different scope, and are mentioned briefly for the sake of completeness only.

An approach to 3D modeling is *Teddy*, for example, by Igarashi et al. It allows for drawing 3D models of freeform objects like stuffed animals [57] by 2D sketches. Another approach to 3D design by sketching is *ILoveSketch* by Bae et al., which allows professional designers to create 3D models [8]. This task is aided with dedicated functionality like mirroring strokes, e.g., to create two identical wings of an airplane by just drawing one wing.

For animation, *Motion doodles* by Thorne et al. allow for first sketching a character similar to a stick figure, and then describing a path this character travels along using one stroke [96]. The shape of this stroke is interpreted to decide when the character goes forward and backward, when to jump, or when to run. Davis et al. [33] also allow for creation of stick figures. Animations are described by sketching annotated key frames of the animation.

Turquin et al. allow for drawing garments interactively. The user draws the front or the back of the garment, and the system automatically dresses a figure with that garment [97]. Igarashi et al. also allow for placing 2D clothing on 3D models by requiring the user to place identical marks on both the clothing and the model [56]. The system then places the clothing on the model by overlapping identical marks.

The group of approaches closer related to our research are those where technical drawings are processed. In the following sections, four approaches are discussed: *GRANDMA* in Section 3.1, *LADDER* in Section 3.2, *Sketch Grammars* in Section 3.3, and *InkKit* in Section 3.4. These four approaches are consid-

ered in detail, because we find them either important and influential, or because their objective is similar to ours. Section 3.5 briefly illustrates a broader selection of further approaches. Section 3.6 draws a comparison between DSKETCH and GRANDMA, LADDER, Sketch Grammars and InkKit.

3.1 GRANDMA

As already mentioned in Section 1.1, the work of Dean Rubine, published in the early nineties, had a great impact [83, 84]. Based on the observation that it is difficult to create a hand-coded recognizer, he tackled the problem of automatically generating recognizers. To this end, a framework called *GRANDMA* (Gesture Recognizers Automated in a Novel Direct Manipulation Architecture) was developed. It allows for automatic generation of recognizers based on training examples of gestures. About 15 examples per gesture are reported to be sufficient for a reliable recognizer. Recognition is based on 13 features which were selected due to some reasonable criteria: small change in the input should result in small change in the features, and for reasons of efficiency the number of features should not be too high, but high enough so that different gestures can be distinguished reliably. For each input stroke each of the 13 features is computed and classified by a linear function that computes the weighted sum of the features. Each gesture has its own set of weights. The stroke is interpreted to be the gesture which yields the greatest weighted sum. The weights are determined by examining the training samples.

Rubine gives two reasons for using single-stroke gestures. First, he wants to avoid the issue of segmentation. Second, single-stroke gestures can be memorized more easily. For the second aspect, the focus on gestures is important. Gestures are meant to represent commands. After a gesture has been processed, its representing stroke is removed from the screen. This is a contrast to other, more current approaches like our own, where the user's strokes are preserved.

Using different sets of gestures, recognition rates exceeding 96% are reached, the higher the smaller the number of different gestures is. Peak values are reached for 30 gestures to distinguish. Even back in 1991, the performance of the implementation was very good, with less than 10ms for the classification of these 30 gestures.

3.2 LADDER

In the course of her PhD, Tracy Hammond has developed a generic approach to sketching called *LADDER* (Language for Describing Drawing, Display, and Edit-

ing in Recognition) [48, 46, 47]. The approach is two-fold. On the one hand, it consists of a high-level description language which allows for an easy specification of what kinds of shapes are to be recognized. On the other hand, the system is equipped with a generic recognizer that is customized by the specification to recognize the described shapes. These two principles are very similar to DSKETCH. However, unlike our approach LADDER completely lacks an analysis stage. The general idea is to provide an easy-to-use, but powerful system to allow experts in a domain to create sketching editors, even if they have only little understanding of the technical aspects of sketching.

The description language can be used to describe the visual appearance of shapes in terms of primitives like *point*, *path*, *straight line*, *curve*, *arc*, *spiral*, *ellipse* or *text*. Recognition of primitives is also guided by constraints, which are either *hard* or *soft*: For successful recognition, hard constraints have to be satisfied, while soft constraints should be satisfied, but do not have to be. Also, the description language allows for the specification of editing operations, and of a beautified visual appearance that replaces the user's strokes after successful recognition. These two aspects are different from DSKETCH, but suit the philosophy of the proposed recognizer very well (see below). Groups of shapes can be defined, but their only use is for editing operations. A possible improvement in terms of recognition rates, driven by a top-down approach, is considered, but is not implemented. The description of shapes even supports abstract shapes and inheritance, inspired by the concepts of object-oriented programming languages. A complete description of the language and detailed examples can be found in the appendix of Hammond's PhD thesis [49]. Two advantages of LADDER are the expressive power of the language and the recognition rates of its recognizer, which are very good even for different domains like Japanese Kanji [92], or constraint network diagrams [50].

Primitives are recognized by the feature-based low-level recognizer of Tevfik Sezgin [91, 89]. Here, for each stroke curvature and speed data are combined to obtain a polyline representation of the stroke. In the second step, segments of that polyline are replaced by Bezier curves to better fit curved parts. Finally, using simple features, primitives are recognized. To assemble shapes from these recognized primitives, a rule-based system is used. The specification is transformed into rules for the system, the recognized primitives are added as facts. From these facts shapes are constructed according to the rules. The complete recognizer works incrementally. Partial findings of the rule system are kept. After new strokes are drawn, new primitives are recognized and added as facts. The system then tries to continue the partial findings with the new facts.

A severe drawback of this rule-based recognition is its runtime performance. Hammond admits in her PhD thesis that the runtime may exceed one hour if there are 30 or more lines on the canvas that have not been used for high-level

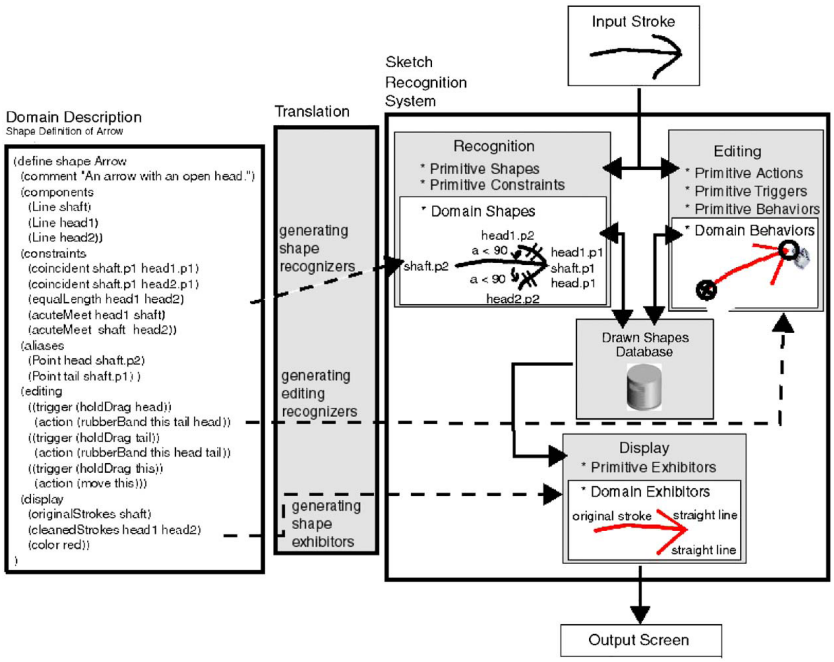


Figure 3.1: Architecture of the LADDER system [48].

shapes [49]. Accordingly, she proposes another recognition approach that combines primitives in a non-deterministic manner. In order to achieve acceptable runtime performance, all primitives are indexed first, in a way that each subsequent access can be performed very quickly. This avoids the extreme runtimes that may happen with the rule based system. The new recognizer now has fully replaced the old one, and is the basis for all further research in her group; however, in many of the group's publications, still the older journal article [48] is cited.

Shapes can be specified to be *final*; whenever a final shape is successfully constructed, the facts representing the shape's primitives are removed. As a consequence, once a final shape is recognized, it is fixed and cannot be undone later, even if there would be a better interpretation. If a shape is not final, its primitives are preserved and can be used again, but this has a negative effect on the runtime. Context is hardly considered; however, the presence of other shapes can be specified to influence the recognition result; for example, a circle can be recognized as a pin joint if it is placed on two closed shapes (the bodies it connects). Due to the eager, incremental recognition it is reasonable to specify editing operations, e.g., moving the head of an arrow, while keeping its tail fixed. If there were no eager recognition, such operations could not be applied. The overall architecture of LADDER is shown in Figure 3.1.

Similar to our experience, the overall recognition rate strongly depends on

the recognition rate of the low-level recognizer. If primitives are missed at this level, no recovery is possible later, the primitives are inevitably lost. In our case, this concerns the transformers (cf. Section 4.1). In the case of LADDER, the feature-based recognizer by Sezgin is affected, and improvements are in the focus of research in Hammond's group. Brandon Paulson suggests a new low-level recognizer called *PaleoSketch* [76], which is also based on features. His main contribution is the introduction of two new features based on the direction graph combined with a deliberate choice of existing features. Supported primitives include *line*, *polyline*, *circle*, *ellipse*, *arc*, *curve*, *spiral*, and *helix*. The basic assumption is that each primitive is drawn in one stroke. If several primitives are drawn by one stroke, the stroke is tried to be split. However, this functionality is very basic in its current state, and is planned to be the subject to future work. The recognition rate of primitives is reported to exceed 98%. However, in their test setting, each subject was required to always draw one primitive in one stroke, with only one exception, where two given primitives were to be drawn in one stroke. This is a very strong restriction, which is not very natural. Nevertheless, PaleoSketch is very promising, and seems like an improvement over previous work. PaleoSketch is integrated in LADDER and serves as an alternative to Sezgin's recognizer.

3.3 Sketch Grammars

An approach driven by formal considerations is that of *Sketch Grammars* (SkG) by Gennaro Costagliola et al. [21, 23, 20]. SkG extends the formalism of *Extended Positional Grammars* (XPG) [29], which itself is very powerful and allows to describe languages from all classes illustrated in the introduction to Chapter 2. SkG allows for describing both the visual appearance of symbols (called *ink grammar*), and the syntactical aspects and semantics of a diagram language (called *language grammar*) in the same formalism. The ink grammar follows the same idea as LADDER. Primitives are combined by relations that allow for precisely expressing how the primitives are related, e.g., where they intersect, or which angle two lines enclose. *Actions*, which are attached to the rules, describe how the attributes of the shapes are computed. Furthermore, relations can also be temporal, e.g., to indicate when a second primitive has to follow its predecessor. This is especially useful for describing editing gestures, which can also be achieved by SkG. Temporal relations are not used by the language grammar. The ink grammar and the language grammar taken as a whole describe a visual language.

For the language grammar it is necessary to describe how shapes can be related to each other. This is accomplished via *attachment regions*, the same concept we also rely on (cf. Section 7.1). These are defined for every shape, e.g., both ends of an arrow, or the outline of a rectangle. Then the language grammar refers

to these regions and describes their connection in terms of relations, which are independent of the visual language[1].

Processing a sketch follows the distinction between ink grammar and language grammar. Three steps are involved, shown in Figure 3.2. The first is the recognition of the primitive shapes, where SkG relies on the SATIN toolkit [52]. In the second step, all primitives are considered as terminal symbols for the ink grammar. As SkG extends XPG, which itself extends string grammars (mainly by adding more general relations between symbols than just concatenation), an LR-parsing approach can used for parsing the input according to the ink grammar, i.e., for recognizing shapes. The result is a set of shapes, and a set of strokes that could not be matched yet. In the third step, the recognized shapes are considered in turn as terminal symbols for the language grammar. As the language grammar follows the same formalism as the ink grammar, the same parsing approach can be used. The parser thus tries to derive the start symbol in a bottom-up manner. However, the result is not a derivation tree, but a forest of trees. Each tree in the forest describes a valid interpretation of the sketch, i.e., an interpretation that satisfies all rules from the specification. In order to compare the trees, ranking values for their roots are computed. These values are ultimately based on confidence values for the primitives, and the relations used when parsing the input in order to recognize shapes.

The full approach works incrementally, and applies eager recognition, which is clearly favored by the authors. Each new stroke is immediately processed to recognize primitives. When parsing the input, the system not only recognizes complete shapes, but also keeps track of shapes that are only partially matched. New primitives are used to try and complete these partial shapes. Having recognized new shapes, the language grammar is considered again, and the resulting interpretation is shown to the user as visual feedback.

Parsing the shapes according to the language grammar means to analyze the shapes and their context, and to generate a syntactically and semantically meaningful interpretation, something that LADDER does not do. The authors of SkG also rely on context to resolve ambiguities. Therefore it has been decided for a grammar-based approach. Similar to SkG, in DSKETCH we also exploit context, but use a different formalism for describing syntax and semantics, and a different parser. What is very appealing in SkG is that both shapes and the language itself (and even editing gestures) are described by the same formalism.

More recent development of SkG involves training [22], error recovery techniques [24] and automatic shape completion [25]. Training is used to determine reasonable values for thresholds involved in primitive shape recognition and pars-

[1]Note that we have a different notion of the term *relation* (cf. Section 7.2), as we use it to describe the actual relations between labeled attachment regions, depending on the visual language.

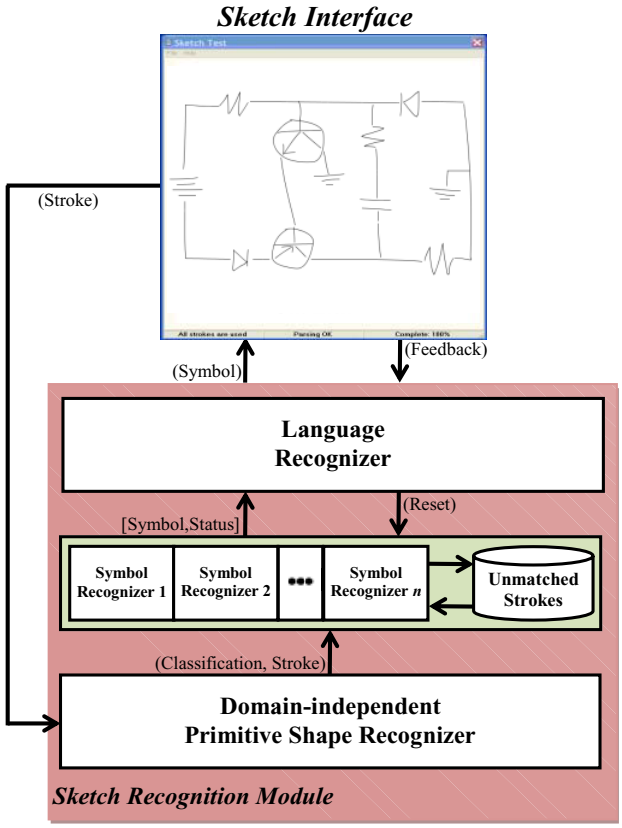


Figure 3.2: Architecture of the SkG system [24].

ing. In user studies, the trained recognizer always outperforms the untrained one. Error recovery is possible because incomplete shapes are also kept. It allows for overcoming missing primitives and incorrectly recognized primitives. Shape completion assists the user in drawing complex shapes by suggesting automatic completion; therefore it is even considered how the user drew such shapes before.

3.4 InkKit

Understanding that sketching is an application domain in itself, Beryl Plimmer and her group at the University of Auckland, New Zealand started working on *InkKit* [19, 79]. It is designed to be a toolkit for creating sketch-based applications. Accordingly, InkKit provides programmers with useful functionality to develop such applications. This includes a generic recognition approach, a GUI, and an easy-to-implement interface to utilize InkKit and embed it in one's own applications. Further topics addressed are beautification [80] and switching between

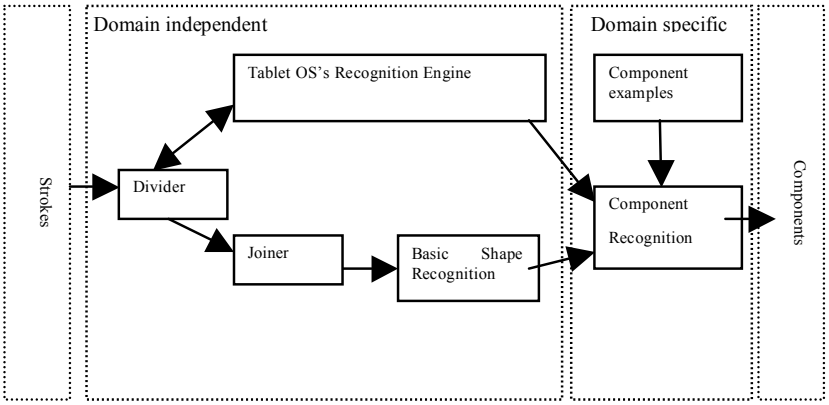


Figure 3.3: Architecture of the InkKit recognizer [79].

different levels of formality [101].

The focus of the research lies more on high-level aspects, and less on low-level technical aspects. For recognition, they basically rely on Rubine's approach, which they have improved in some way. First, the features based on absolute size were replaced by features which only measure ratios. This allows for more flexibility in drawing and is reported to improve accuracy of the recognizer. Second, a *joiner* combines two strokes if both strokes do not form closed shapes and have their end points close to each other. This enables their system to also account for multi-stroke symbols. Using this approach, primitives (which here are called *basic shapes*) are recognized in a generic fashion. The domain-dependent recognition then identifies shapes (called *components*) from these primitives. In order to do so, no source code is required by the programmer, but only examples of shapes. The primitives found in the examples are examined for their relations (intersecting, enclosing, etc.) and their relative positions to each other. Ambiguity can occur among the primitives. For the recognition of shapes, all primitives are first examined for the same relations and positions as in the examples. Then, based on probabilities and likelihood, the shapes are found. The context of a shape (given by other shapes) is not considered for the resolution of ambiguities. Figure 3.3 shows the architecture of InkKit's processing approach. In an informal evaluation of UI diagrams (similar to Section 2.3) the authors claim to reach a recognition rate of about 95%.

Text and graphics may be placed next to each other without any mode change. In order to distinguish the one from the other, the strokes are first passed to a *divider* for recognition. Strokes found to be text are recognized by the text recognizer from Microsoft Windows Tablet PC Edition, which works reliably. All other strokes are considered to be graphics, where shapes are recognized as described above. For the divider, first the one delivered as part of the Microsoft

Tablet SDK was tried, but did not produce satisfying results. As improvement, a own divider has been implemented, which substitutes the Microsoft divider and performs much better. However, the topic is still under investigation [75].

The fundamental goal of the project is to simplify the creation of sketching applications. For this reason, recognition of shapes is not described by source code, but by examples, which are easier to supply, and save lines of code. For the same reason, considerable effort is spent on including more functionality in InkKit to further reduce the amount of domain-specific code. In [39], for example, connectors like lines and arrows, which are very common to many sketches, are examined in three exemplary domains in order to identify certain characteristics. Based on the findings, connectors were integrated into the core functionality of InkKit.

A central aspect of InkKit's user interface is to allow for different sketches from the same domain at the same time. The single sketches can be linked with each other to create one logical unit, which is translated as a whole. The meaning of the sketches, the links between the sketches, and the result of the translation process have to be hand-coded by a programmer who wants to use InkKit within a certain domain. Unlike DSKETCH, no formal approach is taken here which allows for specifying these aspects. On the other hand, hand-coding allows for great flexibility and freedom, thus enabling a broad range of fields of applications. In recent work, the approach has been extended to support even sketches of different domains at the same time [86]. However, there is no formal treatment of syntax or semantics.

3.5 Other Approaches

Besides the four approaches illustrated in detail in the previous sections, there are many other approaches to sketching as well. A selection of these works is given in this section. It contains various approaches on low-level recognition and high-level recognition, which are all guided by different ideas and concepts, e.g., agents, graph transformation, or Hidden Markov Models (HMMs).

Another utilization of graph transformation to sketching (like DSKETCH) is reported by Zanibbi et al. [103]. Using a given set of symbols that have already been recognized, their approach analyzes mathematical expressions. The symbols include numbers, operators, and letters. The sum operator, for example, is given as one symbol; the *equals* token is given as two symbols, both short horizontal lines of approximately equal length. The size and location of each symbol is known. For analysis, a three-step approach is proposed, which is roughly similar to our analysis stage. First, all symbols are examined, and their relative positions to each other are established. Even superscript and subscripts are regarded. The

result is a tree-like structure. Second, tokens comprised of more than one symbol, like equals, or function names, are composed. Tree transformations are used here, which are specified by rules. Third, the resulting structure is transformed into LaTeX code representing the original mathematical expression. Ambiguities like whether a token is above another token, or a superscript, for example, are treated. Ambiguity in the symbols drawn by the user is not considered.

The low-level recognizer *CALI* by Fonseca et al. [38] is based on features, but exhibits two interesting properties. All features are based on the convex hull of all strokes that belong to one shape. As a result, the recognizer is completely invariant to stroke order, direction, or count, but prior knowledge about where individual shapes are located is required. For classification, fuzzy sets are used which allow for a better modeling of ambiguities. The recognizer is not trainable, but only capable of recognizing a predefined set of primitive shapes like triangles, rectangles, wavy lines, etc. The reported recognition rate is about 95%. Using a naive Bayes classifier following [35], instead of fuzzy logic, was also proposed, which can be trained to various shapes. *JavaSketchIt* by Caetano et al. [14] is a high-level sketching system for user interfaces based on CALI. It allows for construction of UI widgets from the predefined primitives by basic spatial and adjacency relationships.

Casella et al. propose a conceptually interesting generic framework for sketch understanding based on agents [16, 17, 18]. They can include arbitrary symbol recognizers, but their organization is quite different from DSKETCH. There is an individual agent for each available kind of shape, called an SRA (*symbol recognition agent*). The SRAs autonomously search for shapes, but may communicate which each other to exchange context information. A central agent may solve conflicts between the SRAs. Use case diagrams from the UML serve as an example. Using a Java-based implementation, good recognition rates are reported. Similar to our findings, this is clearly due to the use of context information.

Agents are also used in *SketchiXML* by Coyette et al. [31, 32]. Their focus lies on user interface design only, as this domain benefits greatly from a sketch-based approach. Their contribution lies in higher-level aspects, e.g., the output format of the system is independent of hardware and operating system, so it supports multiple platforms. Furthermore the behavior of the system can be adjusted to the specific needs of individual users. For recognition, the CALI approach is used.

High-level recognition based on graph-matching is investigated by Lee et al. [64]. Individual shapes are represented as attributed relational graphs which comprise both geometry and topology of the shape. These graphs are learned automatically from training samples. Given that individual shapes have already been located, the corresponding graph for a shape is created first. Then, the best interpretation is chosen by matching this graph with the graphs precomputed for the training samples. Four different approaches to this decision process are com-

pared, all of them guided by a similarity measure. A greedy approach performs best in terms of recognition rate and processing time. The main advantage of the approach is that it is invariant to drawing order or direction.

A flexible approach is suggested by Gennari et al. [40]. It allows for recognition of individual shapes from a complete drawing. The strokes are first approximated by curves and straight lines. Then, using two different approaches, single shapes are located. After recognition of these shapes, e.g., as in [64], using simple scores some shapes are pruned if they overlap with other shapes. Finally, hand-coded rules add domain information. In this article, the domain are electronic circuits. The only assumptions made by the approach are that each shape is drawn one after another, and not interspersed, and that 2 to 14 strokes are used for each shape.

The approach by Grundy et al., *MaramaSketch* [44], adds an additional layer to the Eclipse-based toolkit *Marama* [45] from the same group. Marama allows for the generation of editors for visual languages based on specifications. MaramaSketch uses existing interfaces to extend Marama editors by support for sketched input. The only extra information required are training samples for the shapes. For recognition, the *HHReco* [53] toolkit by Hse et al. is used. Its properties are similar to those of CALI. Multi-stroke input is supported, but segmentation is not possible; instead, it is assumed that the strokes comprising a shape are known. The focus of MaramaSketch lies on higher-level aspects of sketching. Their tool supports (among other things) lazy and eager recognition, informal and formal representation, and explicit user interaction for ambiguity resolution. To a certain extent, even context can be considered for disambiguation, but no details are reported.

Alvarado et al. present an approach called *ASSIST*, which is limited to mechanical devices [3]. Special emphasis is put on ambiguity resolution. First, a recognizer detects primitive shapes like bodies, springs or pin joints. Then, the shapes are scored by some basic rules, which are domain-dependent and hard-coded. The final stage, resolution, selects the final interpretation for the sketch according to the scores. This way, context is considered and ambiguities are resolved. In later work, the group changed their approach to be domain-independent [4, 5] by using the LADDER language to describe shapes, and an incremental high-level recognizer based on Bayes nets instead of the hard-coded scoring schemes. Using Bayes nets, it becomes possible to even identify shapes drawn only partially. However, the approach cannot perform segmentation of strokes contributing to different shapes, and is very limited in its ability to describe context between shapes; syntax and semantics cannot be described at all. Although aimed to be applied in real-time, processing a single stroke takes seconds in many cases, as the Bayes nets grow quickly [2].

The well-known *Back of an Envelope* by Gross et at. [43] extends previous

work of the same group, the *Electronic Cocktail Napkin* [42]. The recognizer is feature-based, clustering of symbols is done by a time-out threshold, and segmentation is not supported. An interesting feature it uses is a 3x3 grid, which records for the grid cells the order in which they were visited by the strokes. A multi-level classification procedure assigns certainty values from 0 to 5 to recognized shapes. To account for automatic resolution of context, *configurations* describe which shapes are related to which other shapes in terms of simple geometric constraints like *contains* or *left-of*. Configurations can also be defined recursively, which allows for a more expressive definition of context. Several examples indicate that arbitrary output is possible as result of the sketch processing, but no technical details are given. Recognition rates are not reported.

Sezgin et al. developed an approach that relies on Hidden Markov Models (HMMs) [90]. It is based on the observation that different users always draw shapes using the same or a very similar ordering of their strokes. This observation is backed by a user study. The advantage of HMMs is that the drawing style of users does not have to be restricted. During training, for each user, and each class of shapes, a HMM is computed. Then the analysis of a complete sketch consisting of many shapes is solved by finding the shortest path in a graph. Reported recognition rates are good, and the performance of the approach excels that of a baseline method using features. Other applications of HMM to recognition are reported by Hu et al. [54, 55], and Yuan et al. [102].

Further approaches to sketching are discussed and compared in [100, 13].

3.6 Comparison

In order to conclude this chapter, and to highlight the value of our contribution, this section compares DSKETCH to those four approaches that have been illustrated in detail before: GRANDMA, LADDER, Sketch Grammars, and InkKit.

Similarities with and differences to other approaches

Common to all five approaches is that they are not tied to a specific domain, but generic. However, GRANDMA has been designed with the goal of gesture recognition, while the other four approaches have been designed for sketches. Accordingly, GRANDMA relies on single-stroke recognition, which suits gestures, while the other four have multi-stroke recognizers which are more convenient when drawing shapes. All five approaches rely on on-line recognition.

LADDER, SkG, InkKit and DSKETCH have been designed with different goals. LADDER is a general-purpose shape recognizer, and its special contribution is the powerful specification language. SkG and DSKETCH aim at under-

standing sketches based on syntax and semantics. LADDER, SkG, and DSKETCH also allow for unrestricted sketching. Finally, InkKit is a toolkit that provides features and methods for building own sketching applications. LADDER, SkG and DSKETCH have in common that shapes are specified using some formalism. In contrast, GRANDMA and InkKit rely on training samples.

Allowing the user to draw freely and unrestricted is fundamental to DSKETCH. As stated in Section 1.2, we have multi-stroke recognition capable of automatic clustering and segmentation. This means that one stroke may contribute to different shapes, or several strokes may contribute to one shape. Furthermore, low-level processing of strokes also considers ambiguity and allows for different interpretations of the same stroke in parallel (cf. Chapter 4). As mentioned before, GRANDMA and InkKit both rely on training samples for their recognizers, which indicates a dependency on the drawing style of the user who committed the training data. However, in this respect InkKit is certainly superior to GRANDMA, and allows for more flexibility, most prominently due to multi-stroke recognition. SkG relies on the SATIN framework for low-level recognition, and is capable of constructing shapes from primitives, which allows for multi-stroke recognition. However, very few details are published.

Except for GRANDMA, all approaches are capable of producing domain-dependent results. GRANDMA, with its focus on gestures, obviously interprets strokes only as gestures. There is no need for any further output. SkG and DSKETCH support the generation of output by two factors. First, as both approaches rely on a parser, derivation structures are generated which express the content of the sketch. Second, these derivation structures describe only syntactically correct sketches, incorrect interpretations are omitted. In contrast, LADDER and InkKit give less support for output generation, and only provide programming interfaces which can be used to access the recognized shapes.

Differences between the approaches can also be observed regarding the issue of runtime performance. GRANDMA certainly is very fast, which is also documented in literature. The perceived performance of LADDER is also very good. Its recognizer works incrementally, which also suggests a good performance. SkG also works incrementally, but no details are published on actual runtime performance. InkKit's performance is also not reported. Regarding our own approach, performance is fast as well, however, not in the range of GRANDMA and LADDER. The evaluation (cf. Chapter 10) shows that this is due to our recognizer, which does not work incrementally. Accordingly, for larger sketches comprising more strokes, runtime increases. However, as DSKETCH applies an analysis stage which, except for SkG, the others do not, performance cannot be compared directly.

Detailed differences between DSKETCH and SkG

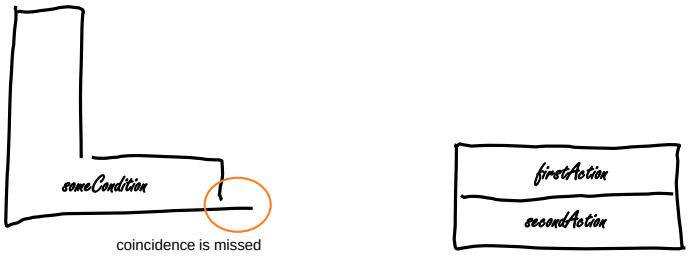
The previous paragraphs show that DSKETCH follows similar ideas like SkG. Differences are in the details. For shape recognition, in SkG shapes are specified with the same formalism as the subsequent parsing process. This simplifies processing. In contrast, shape specifications for DSKETCH are different from the formalism used to describe parsing. An advantage of this is that, similar to LADDER, our shape specifications can be understood and written easily.

The parser used by SkG uses an extension of well-known techniques known from LR parsers for string grammars. On the contrary, we use a hypergraph parser. Our parser solves ambiguities automatically. Therefore, ambiguities (called *conflicts*, cf. Section 6.2) are modeled explicitly. Another interesting detail about DSKETCH and SkG is the use of meta-rules. While we avoid such rules in the parser, SkG prunes search according to two meta-rules. The effect of the application of these rules in terms of recognition rates and runtime performance is not reported.

Detailed differences between DSKETCH and LADDER

An overall difference between our approach and LADDER is the philosophy of the two approaches. LADDER performs an incremental, eager recognition, which is fast in practice. Immediate feedback about recognized shapes can be shown to the user. Context is hardly considered, recognition errors due to the incremental recognition may happen and are accepted. In contrast, in DSKETCH we perform lazy recognition. This allows us to avoid recognition errors as they may occur in LADDER. Accordingly, recognition starts from scratch each time, in order to not miss any shape drawn by the user. Context, considered in the analysis stage succeeding recognition, is very important. By using a search plan, DSKETCH exploits sub-shapes common to different kinds of shapes; in LADDER, this has to be manually specified using the inheritance mechanisms. Common to both approaches is that interspersed drawing is fully supported, i.e., it does not matter if shapes are completed before other shapes are drawn.

Using the example of NSD some limitations of LADDER can be shown, which DSKETCH does not have. Figure 3.4 shows two sketches where LADDER does not produce the desired result. In (a), the loop will not be recognized because lines are not split, so it cannot be determined if the shown endpoint and line coincide. In (b), if statements are specified as final, only one of the three statements will be recognized, as in this case primitives are removed from the recognition process as soon as a shape is found which consists of these primitives. In the case of NSD, this is a severe issue; our user study has shown that no user repeated the strokes dissecting two statements (or other shapes) from each other. If the statements are



(a) (b)

Figure 3.4: Two example sketches for the limitations of LADDER.

not specified as final, runtime decreases; however, too few details are published in order to assess the actual effect.

Conclusion

The most obvious insight gained from related work is that context information is rarely used for disambiguation. Nevertheless, many authors admit its value. Some approaches are domain-specific, and tailored to their domain to employ context. Only rarely, context is formally described in terms of syntax and semantics, which we think is very appealing.

Many other approaches have in common that a more or less severe set of restrictions are imposed on the user. However, recognizers that rely on stroke order, direction, or number, are inconvenient. Even if they may work most of the time, the individual workflow of a user is unpredictable, and should not be interrupted by insufficient processing capabilities. Accordingly, DSKETCH does not rely on the mentioned features, but on abstracted models, where this information is not included.

Although domain-specific approaches can be designed more easily, and may perform better than generic approaches, the contribution of an approach is much greater if it falls in the latter category. Approaches designed for only one domain can assume much stronger requirements, but their impact is obviously limited. Since DSKETCH is generic, we also need a generic mechanism to describe shapes. And as we do not want to rely on features, this mechanism must be an abstract representation of an ideal shape. To this end, we have adopted the idea of SkG and LADDER, where specifications precisely define shapes, as opposed to training samples.

Chapter 4

Preprocessing

This chapter marks the first step in the recognition stage of processing a sketch. Preprocessing is performed by a set of independent preprocessors, which are called *transformers* (cf. Figure 1.4). Input for the transformers are the strokes drawn by the user, output is preprocessed information in the form of *models*. In general, preprocessing (sometimes also referred to as *low-level processing*) is an important step for a sketching system. For several reasons, input strokes always exhibit noise which must be filtered. Noise increases the load on the system and, therefore, processing time. Additionally, later recognition is impeded by noise.

There are different sources of noise. One is *digitization*; a sheet of paper which contains a drawing must be scanned to be digitized. Here, three effects can occur. First, the scanned image is not likely to be monochrome only, even if it is drawn with just one pen, say a pencil. It is very likely that the digitized image will contain several shades of the original color the pencil had. Second, dust and other particles on the surface of the paper will be captured as well by the scanner, meaning that they will be represented as pixels on the image. Third, the scanned lines may have different widths, because a pencil never has the same width all the time when drawing. Because we assume no scanned images, but an on-line sketching approach using dedicated hardware, noise due to digitization cannot occur. However, there are other sources of noise as well. When drawing with a stylus, the user can generate noise accidentally, by slipping off from the stylus, or by pressing buttons on the stylus, thus generating different events. Also, the hardware can fail to track the movement of the stylus, and introduce errors or noise in the data. Because noise due to hardware or due to the user is rather unlikely, we neglect these aspects. Besides, a practical system can solve these issues easily with an appropriate user interface, providing undo/redo functions, for example.

Besides noise, there are further reasons for preprocessing, which apply for DSKETCH. One such reason is to prepare the input data for the recognition pro-

cess by transforming it into a format or representation more suitable for this task. A trade-off between precision and amount of data has to be agreed on. An input stroke is very precise but also rather verbose. Reducing the amount of data for later processing reduces the precision, but decreases the load on the subsequent processing steps.

Another reason for preprocessing is to apply some abstraction. In general, losing precision does not need to be a bad thing. For later processing, abstracted (or aggregated) data can be much more convenient. The challenge is to apply as much abstraction as possible, while not losing important details. As an example consider a stroke that is very straight; it can consequently be represented by its two end points only (given that timing information is of no interest). This means a great reduction in the amount of data, but preserves all necessary details.

To conclude, there are the following reasons for preprocessing: dealing with noise, and getting rid of unnecessary information while preserving essential information. Noise is discarded, as described above. What remains is to apply abstraction, which is described in the following sections. Section 4.1 explains the overall concept and the ideas behind the preprocessing architecture we propose. Sections 4.2 through 4.6 describe how strokes are preprocessed as lines, arcs, links, circles and text. Further work relevant to the topic of preprocessing is given in Section 4.7. Section 4.8 summarizes the chapter.

4.1 Concept

As mentioned in the introduction to this chapter, the input data must be transformed into a format suitable for later processing, i.e., the recognition step performed by the assembler. This step depends on what a stroke is supposed to represent. However, this information is not known initially. Accordingly, the idea is that the preprocessing step does not recognize complete shapes, but primitives only, without any context information or interpretation at hand. Therefore, the input strokes are processed (or *transformed*) in parallel by several transformers, such that *each* possible later interpretation is represented in the transformed data. For example, a stroke could represent a line or a link. Both of these interpretations are generated during preprocessing, and are stored in parallel for the assembler.

In Section 1.2 the four types of primitives we assume have been described: straight lines, arcs, links, and text. Text is processed differently, and will be covered later. When an input stroke is drawn by the user, it is unknown to the system which of the three primitives line, arc, or link it is supposed to represent. Even a combination of several primitives is possible, e.g., a stroke that represents a straight line and an arc connected to the line. Accordingly, each input stroke is processed in parallel by three different transformers: one transformer extracts only

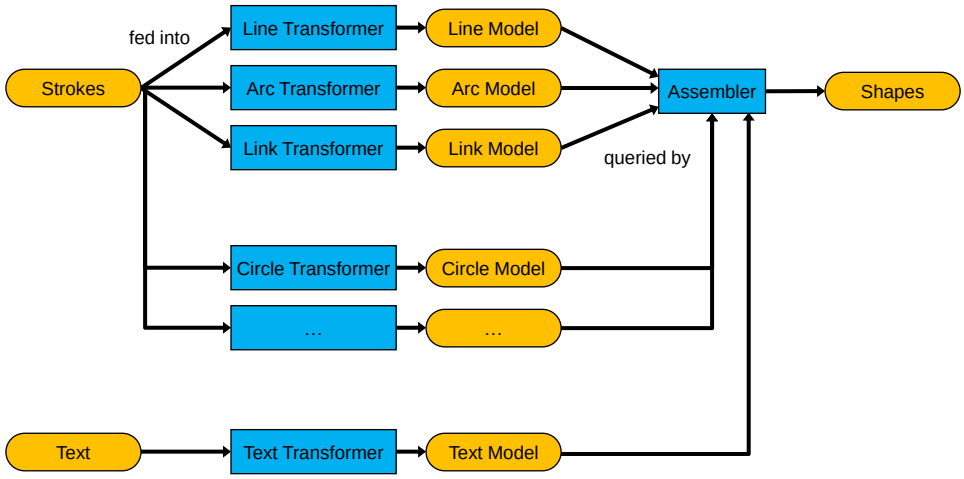


Figure 4.1: Conceptual overview of the preprocessing step.

straight lines, the next transformer extracts only arcs, and the last one extracts only links. The result of each of these three transformations is kept separate from each other. The bottom line is that every transformer induces a certain *view* on the set of strokes drawn by the user.

Figure 4.1 shows this concept in detail. As before, rounded boxes denote data, rectangular boxes denote processing units. Each stroke is passed to the three independent transformers. Each transformer treats the stroke according to its own view on the stroke. The result is stored in respective models. Consequently, a transformer and its assigned model form a logical unit. The contents from the models will later be queried by the assembler (cf. Chapter 5), which yields the shapes. Up to now, each transformer searches for one kind of primitive in the strokes, independent of how the strokes are drawn. However, the proposed concept allows also for integration of transformers which make certain assumptions on the strokes. As an example, a circle transformer is included as well (although a circle is no primitive). For each single stroke representing a circle this transformer identifies the arcs this circle is composed of. The rationale behind this transformer is that circles occur frequently and in different domains, and that this fact can be exploited to improve recognition of arcs. Further transformers (and assigned models) are conceivable, and can be integrated into the shown architecture. This is indicated by the boxes containing dots. Text, the second kind of input data besides strokes, needs no abstraction, as no information can be left out here. Remember, in Section 1.2 we have assumed that text input is indicated by the user in the GUI. Text is fed into a transformer as well, which converts it into a format suitable for the assembler. The following sections discuss each of the five shown transformers in detail, along with their models. Common to all transformers is

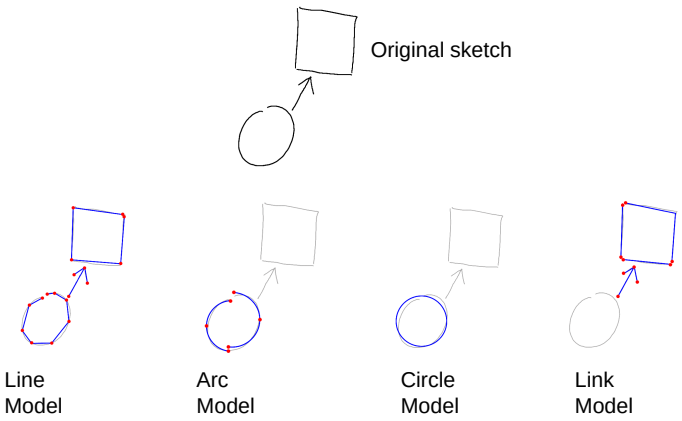


Figure 4.2: A sketch and the four stroke models. The original strokes are shown grayed out.

that references to the original strokes (or text) are stored with the abstracted data. This information is required later (cf. Chapter 6).

An example is given in Figure 4.2. A small sketch of a circle, a square, and an arrow is shown, along with the four models that are generated by the four transformers (since there is no text, the text model is empty). A benefit of the proposed approach is that each model can be updated immediately after a stroke is drawn, which allows for incremental processing. As shown by the models, each transformer is free to discard complete strokes or parts of strokes that do not suit its view. Notice that each transformer works on a best-effort basis, trying to approximate the original strokes as good as possible with primitives. Finally, the same stroke may be represented in different models, which is the case for the circle, for example. This is the most important advantage of preprocessing in DSKETCH: without any prior knowledge what a stroke is supposed to represent, each possible interpretation is obtained and kept in parallel.

4.2 Lines

Basically, the line transformer attempts to map every stroke to a set of one or more straight lines. Basic vectorization algorithms can be applied for this purpose. In order to best suit our needs we have designed a simple heuristic approach, which can be computed very efficiently. Figure 4.2 shows an exemplary sketch consisting of several strokes, and a set of line segments which are the result of the transformation process. As the rectangle and the arrow consist of straight lines only, the identified line segments are a good approximation to the original data. The circle becomes angled, which is no problem. The approximation by straight

Algorithm 1 Transformation of a stroke $s = p_1 \dots p_n$ into straight lines.

```
1:  procedure LINES(s)                          ▷ The stroke s = p_1 ... p_n
2:      L ← ∅                                   ▷ L will contain all straight line segments
3:      a ← p_1
4:      b ← p_2
5:      for i ∈ 3 ... n do
6:          c ← p_i
7:          if |∠abc − 180°| > t_angle then
8:              L ← L ∪ {(a, b)}                ▷ a new straight line segment is found
9:              a ← b
10:         end if
11:         b ← c
12:     end for
13:     L ← L ∪ {(a, p_n)}                      ▷ preserve last straight line segment
14: end procedure
```

lines could be improved if the lines were shorter. However, this is not necessary, as there is no need to closely approximate the circle by straight lines. It is the arc model and the circle model which are supposed to eventually match the circle.

The line transformer processes each stroke independently by splitting it into segments which are nearly straight. To find the samples of the stroke where this splitting must occur, the angles enclosed by three samples a, b, c of the stroke are examined. a, b, c are initialized to the first three samples of the stroke. Then, the angle $\angle abc$ is compared to 180°. If the difference exceeds a threshold $t_{angle} = 25°$, a straight line is found from a to b, and a, b, c are assigned new values. a becomes b, b becomes c and c becomes the subsequent sample of the stroke. If the difference of the angles does not exceed the threshold, then b is replaced by c and c becomes the subsequent sample. Algorithm 1 lists this procedure[1]. Finally, the line model contains a set of straight lines, represented by their end points. Note that the value for the mentioned threshold t_{angle}, as all other values for thresholds, is determined empirically.

The line transformer performs additional tasks which are not shown in the algorithm. To avoid clutter in the line model, those line segments which are very short are discarded. Also, the transformer discards all line segments whose direction does not match at least one line primitive in the specification, as these lines will never be queried later. For example, no shape from the GUI builder uses a

[1]Note that it is theoretically possible to construct a stroke formed like a spiral with increasing radius where this algorithm considers the connection between the first and last sample of the stroke as one straight line, and nothing else. However, this never happens in practice, and does not mean a limitation of the algorithm.

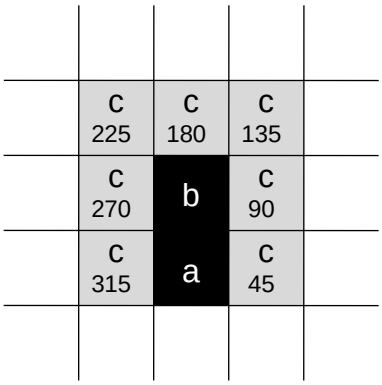


Figure 4.3: Possible angles between three consecutive samples that are too close to each other.

diagonal line (cf. Figure 2.4), so the line transformer discards all diagonal lines immediately. Finally, if lines intersect or have an end point close to another line, they are split; this is necessary and will be discussed in Section 5.4. For the sake of simplicity, splitting of lines is not shown in Figure 4.2.

Because modern hardware allows for very high sampling rates, it frequently happens that consecutive samples of a stroke are received from neighboring locations on the screen, i.e., their distance is 1 or $\sqrt{2}$. Depending on the hardware, this even happens with moderately quick drawn strokes. Only very quickly drawn strokes do not show this effect. The problem arising is that $\angle abc$ gets a multiple of 45°. This renders the whole algorithm useless, as only perfectly straight lines can be recognized as such. An example is shown in Figure 4.3. a and b are assumed fixed in this case, various positions for c are shown. Only for c lying above b ($\angle abc$ is 180°, the difference is 0°) a straight line from a to c can be recognized. As a solution those samples of a stroke must be ignored which are too close to each other. These samples can be filtered out in linear time using another threshold t_{dist}, shown in Algorithm 2. Note that function FILTER must be applied to a stroke before it is processed by procedure LINES (Algorithm 1).

4.3 Arcs

The recognition of arcs – quarters of ellipses as defined in Section 1.2 – is performed by the arc transformer. It relies on the single assumption that arcs are always drawn in one stroke. This stroke can contribute to something else, of course, but there are never two or more strokes forming one arc. This assumption is justified by our user study, where indeed each arc was drawn in one stroke (cf. Section 10.5).

Algorithm 2 Filtering of samples from a stroke $s = p_1 \dots p_n$ which are too close to each other. The first and last sample from s are preserved. The filtered stroke is returned.

```
1:  function FILTER(s)                          ▷ The stroke s = p_1 ... p_n
2:      R ← p_1                                 ▷ R will contain the result
3:      a ← p_1
4:      i ← 2
5:      while i < n do
6:          if EUCLIDEAN_DIST(a, p_i) ≥ t_dist then
7:              R ← Rp_i                        ▷ append p_i to R
8:              a ← p_i
9:          end if
10:         i ← i + 1
11:     end while
12:     R ← Rp_n                                ▷ preserve last sample
13:     return R
14: end function
```

As Figure 4.4 illustrates, each stroke is processed in three steps to identify the arcs it describes. Again function FILTER (Algorithm 2) is applied to the stroke as prerequisite. Then, the stroke is split at its inflection points, such that the resulting sub-strokes are each bent completely left or right, without changes in between (cf. Figure 4.4(b)). Straight line segments in the stroke are filtered out. In order to decide about straightness and bending, similar to the line transformer, each three consecutive samples a, b, c of the stroke are taken, and the enclosed angle $\angle abc$ is computed. If this angle is 180°, the stroke is straight between a and c; otherwise it is bent either to the left, or to the right.

In the second step, sub-strokes are approximated by arcs in the following way. For each two consecutive samples of a sub-stroke the angle enclosed with a reference line is computed. The reference line can be chosen arbitrarily, but must be the same for each angle. As each sub-stroke is always bent to one side only, the change of these angles is monotone. Then, by examining the encountered angles it can be determined which quadrant or quadrants are covered by the sub-stroke. An example for an arc in quadrant 2 is illustrated in Figure 4.5. The horizontal line was chosen as reference line. Accordingly, angles between 90° and 0° fall in quadrant 2. The figure shows for each sample of the stroke the angle enclosed with the subsequent sample and the reference line. It is assumed that the shown sub-stroke has been drawn clockwise. It can be seen that the sample labeled a is the first of the arc in quadrant 2, as its assigned angle is the first below 90°. Likewise, the sample labeled b is the last one of this arc, as its assigned angle is

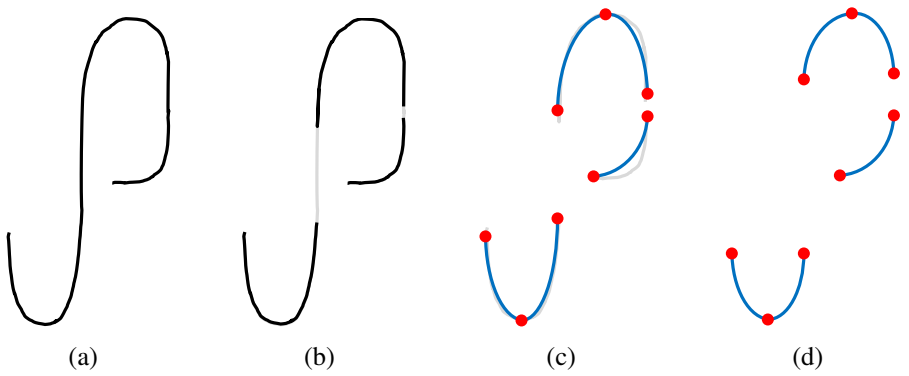


Figure 4.4: Processing of a stroke by the arc transformer. (a) original stroke. (b) sub-strokes completely bent either left or right. The original stroke is grayed out. (c) preliminary arcs superimposed on the sub-strokes. The sub-strokes are grayed out. (d) final arcs after the legs are cut.

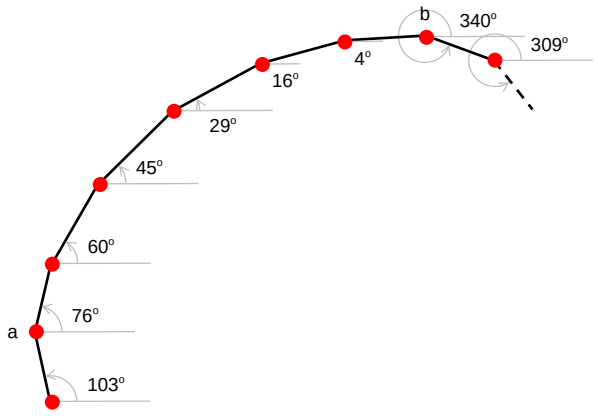


Figure 4.5: Angles enclosed by consecutive pairs of samples (filled circles) of the sub-stroke (bold polyline) with a horizontal reference line. From sample a to b an arc is created, which lies in quadrant 2.

the first one subsequent to a that is not in the mentioned interval. Note that this example serves for illustration only; real sub-strokes show a much higher density of samples, thus the change of the angles is much smaller. The respective arcs which are created for the running example from Figure 4.4 are shown in 4.4(c).

The identified arcs fit the stroke quite well. However, testing has revealed that the legs of the arcs are often very long when a straight line and a consecutive arc are drawn in one stroke, e.g., as in 4.4(a). Hence, the final step for the arc transformer in processing a stroke is to cut these legs. This is done by removing the first and last few samples of the sub-stroke which contribute to the curve of

the arc no more than 5°. Figure 4.4(d) shows the result. All identified arcs are then stored in the arc model by their two end points and the quadrant (1 to 4) they belong to.

4.4 Links

A link is an arbitrarily bent or curved connection between two points. Since this is true for every stroke, each stroke is considered to be a link. The only exceptions are for strokes forming nearly closed shapes like circles or polygons, because the two points connected by the link are supposed to be different from each other. This approach is very simple and efficient. A direct benefit is that intersecting links, e.g., intersecting shafts of arrows, are easily and reliably detected, and not confused. This allows for an unrestricted drawing style for the user. However, this approach also has two flaws: (i) a stroke may contribute to several primitives, not to just one link, and (ii) it may be necessary to cluster several strokes to make one link.

The first flaw mentioned became obvious in the user study. It frequently happened that participants drew arrows in one stroke, i.e., a link for the shaft and straight lines for the arrow head. If strokes cannot be split accordingly, the effect would be that users were forced to always draw arrow shafts in a separate stroke, which is very inconvenient. As a solution, strokes are split at those samples where the local curvature is very high, i.e., where the stroke clearly changes its direction. For this purpose, procedure LINES (Algorithm 1) is applied again, however, using a much greater threshold for t_{angle}, as there is no need for an exact approximation here. Figure 4.6 gives an exemplary arrow drawn in one stroke, which is very similar to the ones from the user study. The arrow head is made by straight lines, while the shaft is supposed to be a link. Consequently, the stroke must be split by the link transformer, in order to properly assign a link to the shaft only, leaving out the head of the arrow. Of course, the head itself is also split two times, as the algorithm does not check for any interpretation (like arrow-head) at this point.

The second issue mentioned above – clustering of strokes to form a link – cannot be solved exhaustively in reasonable time, because the number of combinations between strokes is much too large to be enumerated completely. Hence a heuristics must be applied. For each two strokes having one of their end points close to each other we combine these two, if (and only if) there is no other stroke nearby (cf. Figure 4.7). This process is applied recursively in order to be able to combine more than just two strokes. All other cases of two strokes close to each other, like an end point of a stroke close to another stroke, but not close to one of its end points, or two intersecting strokes, are not combined. In Figure 4.7, (a) is not combined because it is a stroke forming a closed shape; (b) is not combined

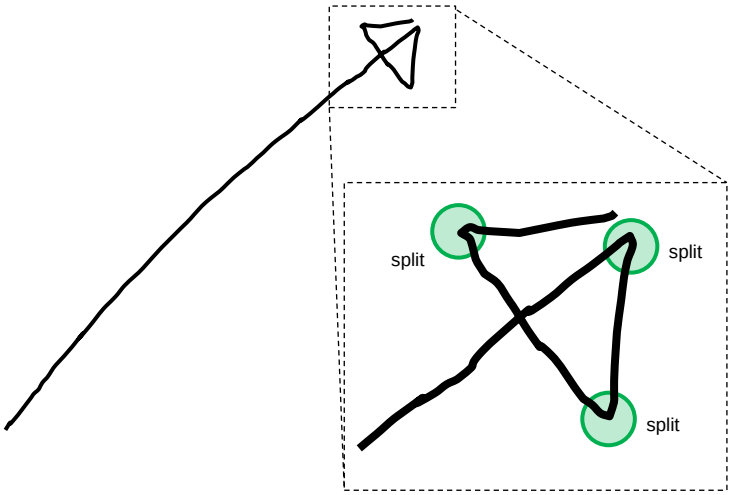


Figure 4.6: An arrow drawn in one stroke. The shaft is supposed to be a link, so the stroke has to be split at the indicated positions.

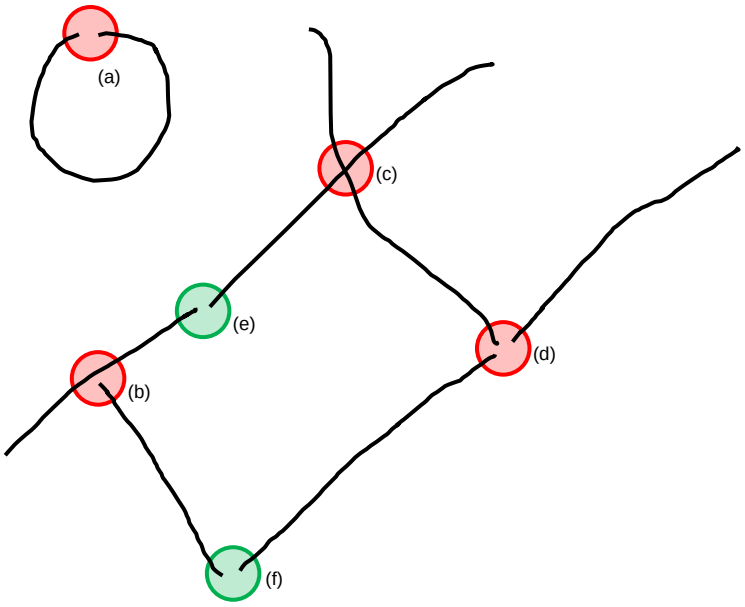


Figure 4.7: Different cases of two strokes being close to each other. (a)-(d) are not combined, while (e) and (f) are.

because for one of the strokes that are close to each other, it is not an end point; (c) is not combined because the strokes are intersecting; finally, (d) is not combined because there are end points of three strokes close to each other, so for every two of them there always is another stroke nearby. (e) and (f) are combined.

Actually, combining strokes is not performed by the link transformer, but will be done on demand later, when the assembler queries information from the link model. The reason is that there is a considerable high number of possible combinations of strokes (even with the mentioned restrictions), most of which will never be queried at all. Hence, it is not reasonable to precompute all of these possible links in the transformer, but rather to do so on demand from the assembler. Just like the line model, in the link model only the end points of links are stored. Because the original strokes are also stored, it is possible to decide whether two links may be combined when queried by the assembler.

4.5 Circles

Circles are composed of four arcs. Therefore, the arc transformer already detects circles. Nevertheless, the addition of a circle transformer is a worthwhile extension. As circles are common to many different diagram languages (for example, from the six diagram languages mentioned in Chapter 2, five use circles), using the circle transformer the reliability of preprocessing step can be increased. Besides, the addition of this transformer (and its model) is an example for how two different transformers can process the same primitives.

In contrast to the other transformers, a feature-based approach is suitable for circles. Based on the assumptions that (i) each circle is drawn in exactly one stroke, and that (ii) this stroke does not contribute to any other primitive, several features of each stroke can be computed and evaluated. Both assumptions hold for each circle drawn in the user study as described in Section 10.5. The implication of (i) and (ii) is that the circle transformer must neither cope with segmentation nor with clustering, which both pose problems for feature-based recognition. The following features are considered:

- Length l of the stroke.
- Center point $p = (x, y)$, radius r and accumulated angle γ. These four values are obtained by a parameter estimation (described below). γ is defined as the angle computed by accumulating the absolute difference between each three consecutive samples of the stroke and 180° (see Figure 4.8).
- Bounding box b (with height h and width w).

First, the parameters x, y, r and γ of a possible circle have to be estimated based on the stroke s. A simple solution would be to compute the average coordinates of all samples from s to find x and y. Then, r gets the average Euclidean distance of each sample from (x, y). γ can be computed according to its definition, i.e., by accumulating the absolute difference between each three consecutive

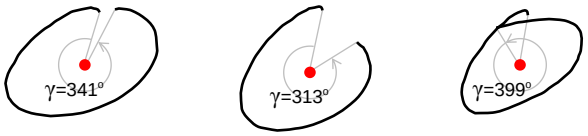


Figure 4.8: Examples of a stroke and accumulated angle γ. The value of γ has no upper bound and can therefore exceed 360°, as in the example on the right.

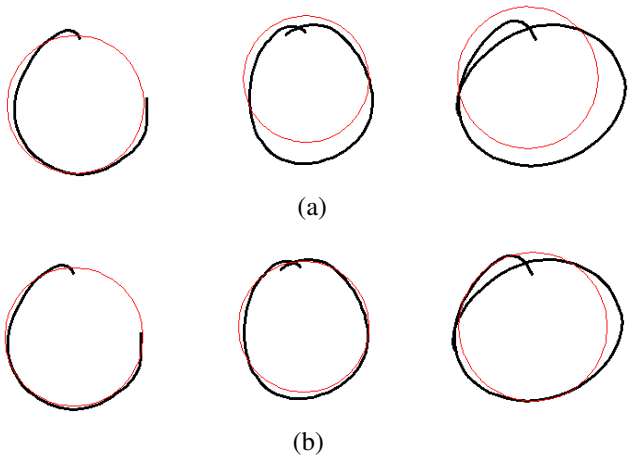


Figure 4.9: Strokes and superimposed circles. (a) the center of the circles is computed as average of all samples. (b) the center of the circles is computed as average of all intersections of perpendiculars going through the midpoints of the chords.

samples of s and 180°. Note that this feature is identical to the *total rotation* as in [76], or the *total angle traversed* as in [83].

The problem with this naive approach is that the values computed for x and y often are not exact if the stroke does not describe a good approximation to a circle, e.g., because it was drawn very sloppy. This issue is revealed in Figure 4.9(a). It shows three examples of strokes and superimposed circles obtained from computing the mentioned average values. The circles do not approximate the stroke well. Consequently, a different approach has to be taken. r and γ are computed as described above, but x and y are not. Instead, (x, y) is calculated as the point with the minimal distance to all perpendiculars to all chords (given by two consecutive samples from s). In case of a perfect circle or arc, all of these perpendiculars intersect in exactly one point. Otherwise, there can be usually seen a cluster of individual intersections. The average of all points of intersection is taken for (x, y). Figure 4.9(b) shows the same examples as before, but with the center of the circles computed in the improved way.

Given the estimated parameters, deciding whether a stroke may be reasonably interpreted as a circle is easy. Since small circles tend to be drawn more sloppily,

Algorithm 3 Transformation of a stroke s into a circle.

```
1:  function CIRCLE(s)                ▷ s is the input stroke, the result is (x, y, r) or nil
2:      (l, x, y, r, γ, h, w) ← COMPUTEFEATURES(s)
3:      if not b contains (x, y) then      ▷ bounding box contains center of circle
4:          return nil
5:      end if
6:      if γ < 300° then
7:          return nil
8:      end if
9:      if w ≤ 40 and h ≤ 40 then                    ▷ check if the stroke is small
10:         if w > 3 · h or h > 3 · w then   ▷ bounding box not too high or wide
11:             return nil
12:         end if
13:         l′ ← γ/180° · r · π        ▷ length of a perfect circle with accu. angle γ
14:         if l′ < 0.75 · l then   ▷ the stroke must not be too long compared to l′
15:             return nil
16:         end if
17:     else                                          ▷ the stroke is not small
18:         if w > 2 · h or h > 2 · w then
19:             return nil
20:         end if
21:         l′ ← γ/180° · r · π
22:         if l′ < 0.9 · l then
23:             return nil
24:         end if
25:         if HASSHARPBENDS(s) then    ▷ ckeched only for strokes not small
26:             return nil
27:         end if
28:     end if
29:     return (x, y, r)
30: end function
```

they have to be distinguished somehow. We define that a stroke is *small* if neither height nor width of the bounding box exceed a value of 40 (another threshold determined empirically). Then, the decision whether a given stroke is a circle is made by function CIRCLE (Algorithm 3), which computes the triple (x, y, r) to indicate that a stroke is a circle, otherwise **nil**. If a stroke is found to be a circle, the features x, y and r are used to describe the circle and are, thus, stored in the circle model.

In the algorithm, the features described above are computed first (cf. line 2).

Then, evaluation takes place. For small circles, the ratio between width and height of the bounding box is usually not as close to 1 as for larger circles (cf. line 10). The same holds for the difference of the actual stroke length l and the length of a perfectly precise circular line with radius r and accumulated angle γ (cf. line 14). Finally, if the stroke is not small, we search for sharp bends in the stroke, which are not supposed to occur for a circle (cf. line 25). A sharp bend is defined as three consecutive samples of the stroke enclosing an angle less than 110° or greater than 250°. Again function FILTER (Algorithm 2) is applied for this measure to be applied properly. The measure is necessary to distinguish circles from squares, which otherwise are sometimes misinterpreted as circles, too. For small strokes, this measure does not yield valid results, as the described angle easily becomes too sharp for three samples, even if the intention was to draw a circle.

4.6 Text

As stated in the introduction to this chapter, text is handled completely different from strokes. This means that the user, while drawing the diagram, has to indicate when text is entered and when graphics are drawn. The way this indication is done depends on the user interface. An alternative is to automatically separate text from graphics, which is done by a processing unit called a *divider*. However, this is a very challenging issue which is neither discussed nor solved in this thesis, but tackled in related work such as [75, 10]. Using a divider, the system architecture shown in Figure 4.1 would change to the one shown in Figure 4.10. One way or another, the models would contain the same data (given that the divider works reliably). Hence there is no effect on subsequent processing steps.

Manually indicating the entering of text has the advantage that special hardware can be used (e.g., a regular keyboard, or *Thumbscript*[2]), or approaches that require a special GUI (e.g., an on-screen keyboard, menu-augmented soft keyboards [58], *Fitaly*[3], *SHARK* [104], *Quikwriting* [78], or *Cirrin* [66]). It remains an open question what is preferred by users, and what approach or approaches lead to the most reliable input. A further investigation of text input approaches for sketching systems is given in [88].

Since text is not directly written on the canvas, is has to be rendered using some default font. Text is assumed to be written always horizontally, and so is the space it requires assumed to be rectangular and axis-parallel. The size of this rectangle depends on the text, the font face, the size of the font, and the font style (for example, italic or bold). The only thing the text transformer therefore

[2] see http://www.thumbscript.com/

[3] see http://www.fitaly.com/

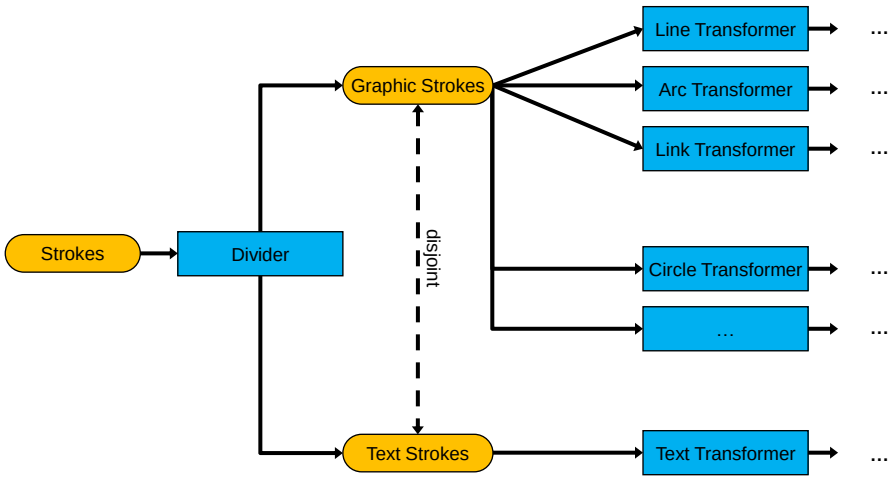


Figure 4.10: Conceptual overview of the preprocessing stage if a divider would be used.

computes is the location and the size of this rectangle, using the text as input, based on the current font. This information is stored in the text model.

4.7 Future Work

There are different aspects of future work concerning the preprocessing stage. In Section 4.6 the use of a divider is discussed. The user interface of a sketching system could benefit from such an approach, but the concept as well as possible gains need to be further investigated.

In a diploma thesis the recognition of hatched and filled regions in drawings has been tackled, with some very good results reported [94, 95]. Hatched regions, as they occur in architectural diagrams, for example, are recognized by transforming input strokes into a parameter space, where characteristic patterns can be identified more easily. For the recognition of filled regions, like transitions and tokens in Petri nets, for example, three features have proven to be sufficient. An open question is how best to integrate this component into our transformer-model approach. For strokes representing hatched or filled regions, the transformers described so far may produce much clutter, which can hinder the actual shape recognition. This is especially true for the line transformer.

Similarly, dashed and dotted lines frequently occur in diagrams. For example, a dashed arrow is used to indicate that a class is implementing an interface in a UML class diagram. Recognition of such lines is still an open issue. Finally, other approaches to low-level processing like [76] may be included as well, either in addition to the proposed transformers, or as replacements.

4.8 Summary

In this chapter, our approach to low-level processing in DSKETCH has been described. The key idea is to generate different interpretations of each stroke in parallel. There is no context information available at this point to decide about a reasonable interpretation. Every transformer works on a best-effort basis to process a stroke according to its own view. In doing so, the information conveyed by the strokes gets abstracted to a certain extent, which is an effective way to deal with noise and imprecision. Each transformer is free to discard complete strokes or part of strokes if they happen to be outside of its view.

For each of the three graphical primitives (lines, arcs, links) there is an independent transformer (and model), and there are two further transformers, one for text and one for circles. Additional transformers, either for new graphical primitives, or for special shapes which exhibit a complicated structure, can be integrated easily. Each model is updated immediately after a stroke is drawn.

Of course, approaches to low-level processing have been published before, as has been illustrated in Chapter 3. Examples are PaleoSketch [76], the recognizer by Sezgin [91], and the SATIN framework [52]. The transformers we have presented in this chapter are an alternative to those approaches. There are two points in favor of our solution:

- The transformers are very efficient and take virtually no time, as the user study shows. In fact, the average time to transform a single stroke by all transformers ranges below 0.5ms (cf. Section 10.1).

- Unlike other approaches, our transformers are completely independent; the consequence is that each transformer can split each stroke differently, and therefore can approximate the stroke more precisely. Other approaches often do not do so, but split the stroke at the same points for all primitives.

Nevertheless, the concept of the preprocessing steps allows for integrating low-level recognizers from other groups as well, by wrapping them into additional transformers.

Chapter 5

Recognition

As mentioned in Chapter 1, recognition is one of the most challenging issues of every approach to sketching. In our architecture, shown in Figure 1.4, the *assembler* is the central component that performs the actual recognition. Based on the contents of the models obtained by preprocessing, the assembler combines single primitives into complex shapes. Therefore, the assembler has to query the models. The key idea is that the assembler treats every model in the same way. It has no specific information on any model regarding what primitives are stored in this model. Accordingly, the assembler always queries each model for a primitive. This is crucial to the integration of new transformers and models, as it requires no changes in the assembler at all.

The actual recognition process is discussed in Section 5.4. Before, two prerequisites have to be explained. The first is how queries of the assembler to the models look like, and how the results to these queries are determined. This is illustrated in Section 5.3. The other prerequisite is the *search plan*. Its purpose is to determine the order in which primitives are searched for by the assembler. On the one hand, the search plan guarantees that the single primitives of a shape are connected to each other as indicated by the specification of the shape. On the other hand, the search plan determines the search order in such a way that primitives shared by different shapes are searched for conjointly. This improves performance. The search plan is discussed in Section 5.2.

When specifying shapes, the expressive power of primitives alone does not suffice. For example, using only primitives it cannot be specified that the width of a rectangle should be greater than its height. To add expressive power, *constraints* are used in addition to primitives. Section 5.1 describes constraints, and how shapes can be specified using these constraints and primitives.

Section 5.5 discusses an approach to *rate* shapes. This is necessary in order to argue about the quality of shapes. The final two Sections 5.6 and 5.7 of this chapter deal with future work and provide a summary.

Figure 5.1: Examples of shapes which are not connected.

Sections 5.2 through 5.4 assume that all primitives of a single shape are connected. However, many practical shapes fail to satisfy this assumption. Two examples of such shapes are shown in Figure 5.1. In order to also recognize such shapes, each connected part is recognized independently. Then, the complete shape is assembled from the parts. Constraints can be used to describe how the parts should be related to each other. An example is given in the next section.

5.1 Constraints and the Specification of Shapes

It is obvious that shapes are composed of primitives. A rectangle, for instance, is composed of two vertical straight lines and two horizontal straight lines. This issue has already been discussed in Section 1.2; examples have been given in Figure 1.5. Recall that there are junction points where primitives are connected to each other, and simple points where primitives are not connected. In the case of the rectangle, there are four junction points, one at each corner. Whereas this example is straightforward, others are not. An open-headed arrow, for instance, may consist of three straight lines, two for the head, and one for the shaft. All three lines meet in one common junction point, which is the tip of the arrow. Figure 5.2 gives examples of lines which satisfy this description. Obviously, primitives and their incidence do not suffice to completely specify arrow heads. It is also necessary to limit the length of the lines of the arrow head, require them to have similar lengths, require the shaft to have a considerably greater length, and enforce certain angles between the three lines meeting at the tip. This is achieved by constraints. Common to all constraints is that they are solely based on points.

While a great number of constraints are conceivable, we found a relatively small set to be sufficient for most cases. The following enumeration gives all of these constraints. There are constraints for

- Comparing the angle described by three points to either a fixed value, or to another angle, described by three other points. Comparing means either *equal*, *not equal*, *greater than*, or *less than*.
- Comparing the distance between two points to either a fixed value, or to another distance between two points. Comparing means again either *equal*, *not equal*, *greater than*, or *less than*.

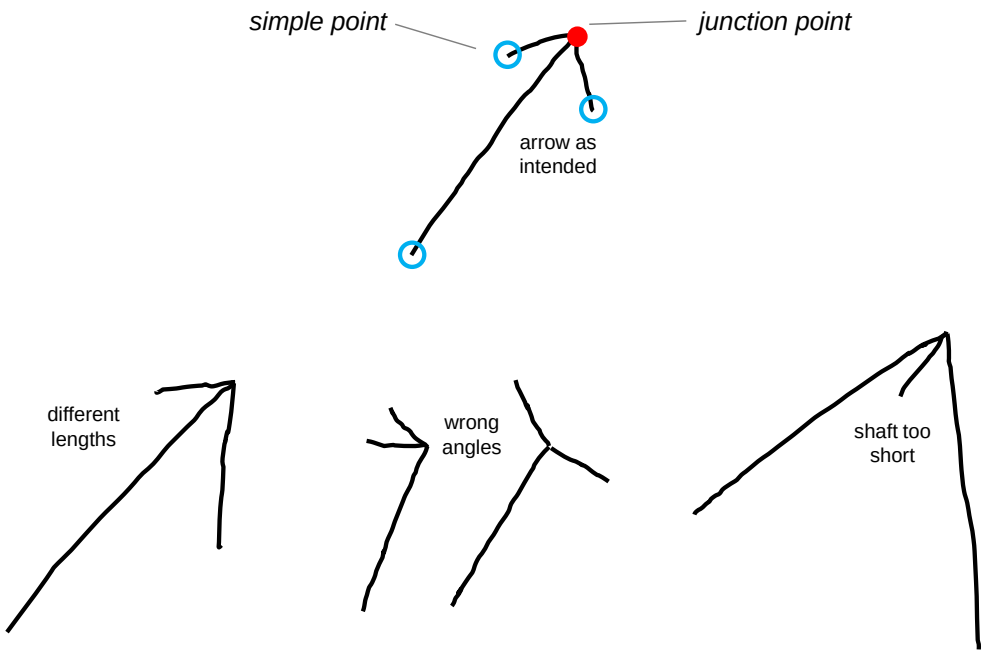


Figure 5.2: An arrow as intended, and examples of possible shapes with the same primitives if no constraints are in place.

- Comparing the direction of a thought straight line between two points to a given value (horizontal, vertical, ascending, or descending). In this case, comparing means only *equal* or *not equal*.

- Comparing the coordinates (x or y) of a point to those of another point. Comparing means again either *equal*, *not equal*, *greater than*, or *less than*.

Of course, when comparing values as indicated by these constraints, thresholds have to be applied to provide for the impreciseness of hand-drawing. It is highly unlikely that two angles are ever equal, for example. However, the specification always describes a perfect shape, so it is reasonable to describe comparison in terms of *equal*, or *not equal*. The mentioned lack of precision is handled by the assembler. Further constraints are conceivable, but are not needed to specify the diagram languages described in Chapter 2. Anyway, further constraints can be easily integrated.

Just like LADDER (cf. Section 3.2), we distinguish between *hard* and *soft* constraints. For hard constraints, the assembler checks that these are satisfied for each shape that is recognized, and discards all shapes where this is not the case. Soft constraints are only used to influence the rating of a shape (cf. Section 5.5). A shape is not removed just because a soft constraint is violated.

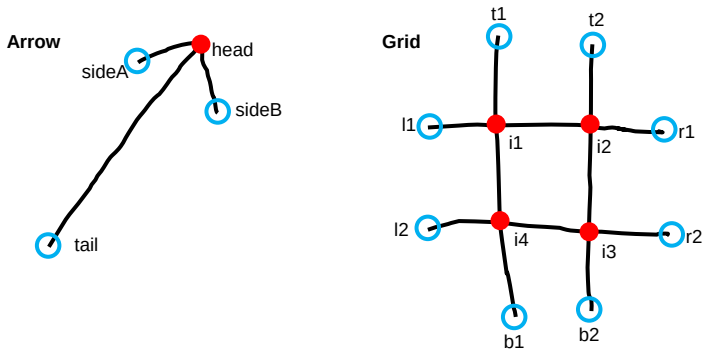


Figure 5.3: Examples of shapes. All junction points and simple points are named.

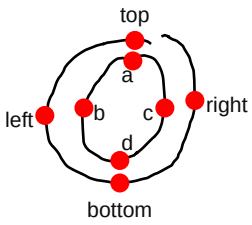
The specification of a shape has to include (i) its primitives, (ii) constraints on these primitives, and (iii) *attachment areas*. The latter will be covered in Section 7.1. To specify (i) and (ii), each distinct point of a shape is uniquely labeled. Two different examples of shapes are shown in Figure 5.3. All points, either junction points or simple points, are labeled. Constraints are not explicitly shown; they are rather represented by the visual appearance of the shapes. Details of the textual specification of these two shapes, including the constraints, can be found in Appendixes A.1.4 and A.6.3. Note that the textual representation is sufficient. The graphical representation shown here serves for clarity only. Also note that text is the only primitive which can be specified to be *optional*, which means that it may or may not be present. This allows for more flexible specifications.

The shape shown in Figure 5.4(a) is not connected; it comprises two circles, one completely contained in the other. As mentioned in the introduction to this chapter, such shapes are recognized by recognizing each connected part first, and then combining these parts. Given the specifications listed above, it can be expressed that the one circle must contain the other one (cf. Figure 5.4(b)). Note that a prerequisite to the combination of shape parts is that no sub-stroke is used in two different parts. In this example, a sub-stroke must not be used in both circles.

For the specification of an actual diagram language, all its shapes have to be defined in terms of (i)–(iii). Based on these specifications a search plan is computed, as described in the next section.

5.2 Search Plan

The actual recognition of shapes, described in Section 5.4, collects the single primitives of a shape, and then assembles these primitives to the complete shape. A search plan dictates the order in which the primitives are searched for. The



(a)

top.y less than a.y
left.x less than b.x
right.x greater than c.x
bottom.y greater than d.y

(b)

Figure 5.4: Example of an unconnected shape and its constraints. The origin is in the upper left corner.

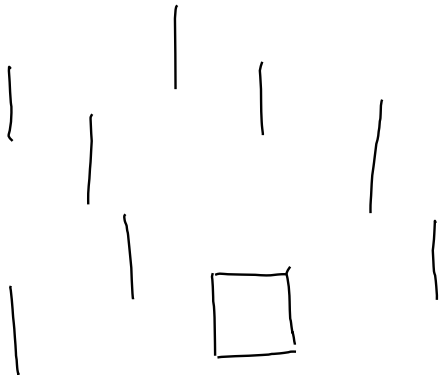


Figure 5.5: A rectangle and many vertical lines hampering the recognition.

reason is that (i) certain primitives require other primitives to be identified first, and (ii) each primitive which is found has to be connected to some other primitive which has been found before. An example for (i) are text regions, which are constructed as soon as all of the necessary points are identified. An example for (ii) is shown in Figure 5.5. A rectangle is searched. It consists of two vertical and two horizontal lines, connected at their end points. If the recognition process first identifies the two vertical lines, then trying to identify horizontal lines connecting these vertical lines at their end points, many pairs of vertical lines are considered in vain. Searching for a vertical line, and then for a horizontal line connected to that first vertical line is the better option here, because it does not require to try each combination of two vertical lines.

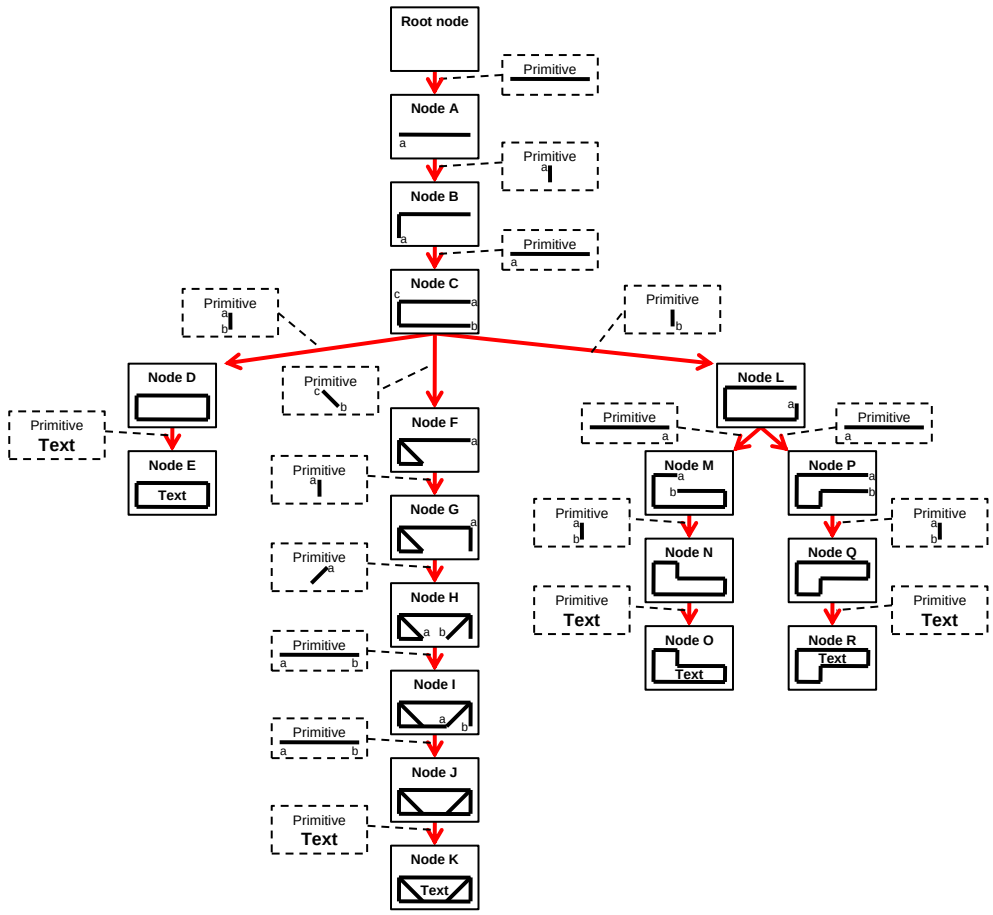
Another motivation for computing a search plan is to determine when to search conjointly for the same primitives occurring in different shapes. Depending on the specification of the diagram language, different shapes may contain the same primitives. Examples are Nassi-Shneiderman diagrams (NSDs) (cf. Section 2.2). In the specification, each of the four shapes contains a horizontal line. Additionally, in each case there is an additional vertical line attached to the left end point of

Figure 5.6: An example for shared primitives among the shapes of NSD. The shared primitives are shown as solid lines. The choice of shared primitives is not unique in general.

the first horizontal line, and going up (cf. Figure 5.6). Alternatively, there is also an additional vertical line attached to the right end point of the first horizontal line, also going up. Shared primitives like this can be searched for conjointly, which saves time for the recognition. This example also shows that there are several possibilities for shared primitives in general. In the special case that two shapes have the same visual appearance, using this approach they can be identified at the same time. In Petri nets, for example, both places and tokens are represented as circles. Whenever a circle is recognized, both a place and a token are found. On the other hand, diagram languages like Tic-tac-toe do not have a single shared primitive among their shapes (cf. Section 2.6). In general, the more shapes a diagram language consists of, the more likely shared primitives will occur.

A search plan is a tree where each node represents a (partial) shape. The transition from one node n to the next node n' is done by adding one additional primitive to the partial shape represented by n such that all primitives of the (partial) shape represented by n' are still connected. Each leaf represents a fully identified shape. Additionally, inner nodes may also represent completed shapes (depending on the specification). The root of a search plan is an empty node which has no primitives. The first primitive which is then identified obviously cannot be connected to any other primitive. In the ideal case, each node has no more than one child. This means that all partial shapes represented by this node have one single shared primitive which can be used to continue the search for all of these partial shapes. If such a primitive, shared by all partial shapes, does not exist, the search plan node has more than one child, because in this case the search process must be continued with different primitives.

Figure 5.7 shows an example search plan for NSDs. Node `C` has three children, node `L` has two. This means that there is no fourth primitive shared by all shapes (node `C`), and no fifth primitive shared by the shapes *While* and *Until* (node `L`). At nodes `E`, `K`, `O` and `R` the different shapes are completely identified, which can be also seen in the table. This search plan does not show an example where an inner node represents a completely identified shape. Note that nodes `D` and `L` are different, although they both add a vertical straight line to the partial shape in node `C` at the point labeled b. They are different because in node `D` the horizontal lines are connected by the new vertical line, while in node `L` they are not.



Node	Partial	Complete
Root	*(all)*	
A	*(all)*	
B	*(all)*	
C	*(all)*	
D	Statement	
E		Statement
F	Condition	
G	Condition	
H	Condition	
I	Condition	

Node	Partial	Complete
J	Condition	
K		Condition
L	Until, While	
M	Until	
N	Until	
O		Until
P	While	
Q	While	
R		While

Figure 5.7: One possible optimal search plan for NSD. For each node the partially identified shape is shown. Transitions from one node to a child are triggered by adding the respective primitive, connected at the points indicated by lowercase letters. The table shows for each node which shapes are partially or completely identified.

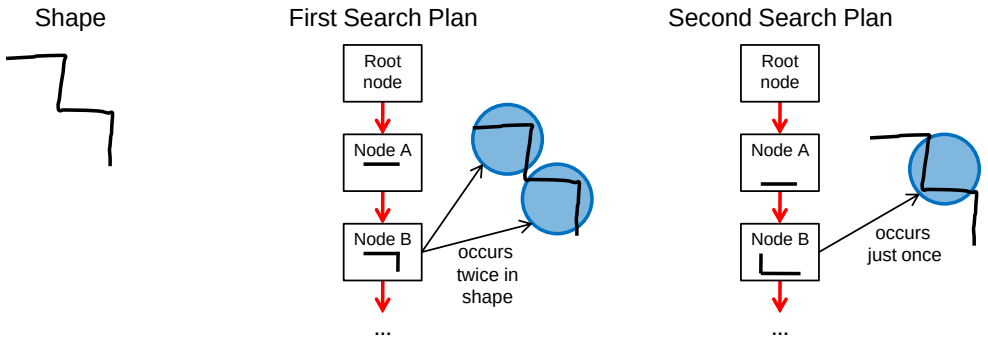


Figure 5.8: Two different search plans for the same shape. The right one is better than the left one, as the partial shape represented by node B occurs just once in the shape, instead of twice.

A search plan is not unique in general. This raises the question which of two search plans is better? The intention is that the search plan increases the performance of the recognition process by demanding search for shared primitives first. Accordingly, the best search plan is the one where recognition takes least time. However, because actual performance depends not only on the search plan, but also on the diagram language, on the sketch, and on the drawing style (cf. Chapter 10.5), it can only be estimated which one will be better when the search plan is computed. Therefore, we define a search plan to be optimal if its number of nodes is minimal. In this case, as much shared primitives are searched for conjointly as possible.

Although the computation of an optimal search plan can be performed exhaustively, we rely on a greedy approach to make this computation more efficient. For each node n representing a partial shape, a new node is created for that primitive which preserves the most alternatives in terms of shapes which can still be reached from n. In case that this criterion is not unique, that primitive is chosen which preserves the least alternatives for the same shape (if this is still not unique, the choice is non-deterministic). In order to explain this secondary criterion, consider Figure 5.8. Two different search plans are outlined, both for the same shape. Of course, both plans show the same number of nodes to fully identify the shape. However, the second one is better, as the partial shape yielded after two steps is unique for the shape, which means that it will not occur as often as the partial shape yielded by the first search plan after two steps. This means that, using the first plan, half of the partial results recognized after the second step turn out as dead ends, which is a waste of computing resources. Using the secondary criterion as described above, the greedy algorithm computes the second plan.

For the computation of a search plan, shown in Algorithm 4, each node is

Algorithm 4 Computation of a search plan from a set S of shapes. Return value is a node n.

```
 1: function SEARCHPLAN(S)
 2:     n ←NEWNODE
 3:     n.partial ← {s|s ∈ S, s not yet completed}
 4:     n.complete ← {s|s ∈ S, s completed}
 5:     S ← n.partial
 6:     while S ≠ ∅ do
 7:         (p, S′) ←SELECTBEST(S)        ▷ select best remaining primitive p
 8:         n′ ←SEARCHPLAN(S′)            ▷ recursively compute child node
 9:         ADDCHILD(n, p, n′)                              ▷ add child node
10:         S ← S \ S′
11:     end while
12:     return n
13: end function
```

treated separately. A node has two disjoint sets that cannot both be empty (cf. the table in Figure 5.7). One set contains the shapes for which the node represents a partial finding; the other set contains the shapes which are completely identified at this node. The root node represents all shapes as a partial finding, as it does not contain any primitives. For a given set of shapes S, the algorithm first selects that primitive p according to the two criteria described in the previous paragraph (line 7). p must not have been selected before, as in this case, p would occur twice in the search plan. Let S' be the set of shapes that share p. Second, a new node is created recursively for the shapes in S' (line 8). This node is added as a child (line 9). If there are other shapes which do not use the primitive p, the process is repeated until every partial finding is represented in a child node. The function SEARCHPLAN is initially called with the set of all shapes of the diagram language, and yields the root node of a search plan.

5.3 Querying the Models

Recall that the assembler queries the models for primitives. Before the details of this process are described in detail in the next section, this section first explains what kinds of queries are possible, and what the results of the queries look like.

There are four kinds of primitives queried by the assembler: lines, arcs, links and text. We discuss each of them separately. When querying for a straight line, the assembler optionally specifies a start point, an end point, and a direction. If specified, the direction (assumed from the start point to the end point) serves as a filter discarding all lines from the result that have another direction. It can be

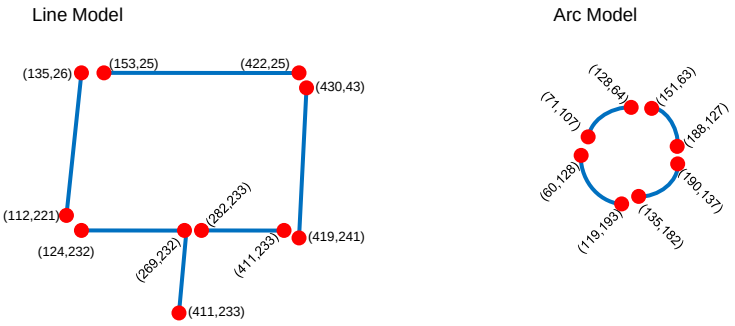


Figure 5.9: A line model and an arc model.

one of eight values: up, up left, left, down left, down, down right, right and up right. If the start point is specified, all returned lines must start in this point. Likewise, if the end point is specified, all returned lines must end in this point. Whether start point and end point are specified is given by the search plan. For example, the transition from node `A` to node `B` in Figure 5.7 requires a horizontal line going down from the point represented by `a`. The end point is not specified in this example.

For arcs and links there is a start point and end point as well, which works exactly the same way as for lines. However, instead of specifying a direction, for arcs both a quadrant and the orientation (clockwise or counter-clockwise) must be specified. Quadrant and orientation are not optional. Links do not have any other special attribute, start point and end point are sufficient here.

In Figure 5.9 a line model and an arc model are shown. The line model stores six lines, while the arc model stores four arcs. The indicated coordinates correspond to the end points of the lines and arcs. Note that both models only contain perfectly straight lines, or perfect arcs, respectively. This is due to the abstraction applied by the transformers, described in Chapter 4.

Now consider some queries to these models. If the line model is queried for a line going up, starting at point $(419, 241)$, it returns the line ending in $(430, 43)$. Although this line is not perfectly vertical, it is a valid result; the models have to respect the lack of precision due to the nature of hand-drawing. Likewise, when the line model is queried for a line going left from $(419, 241)$, it returns two answers: the line ending in $(282, 233)$, and the line ending in $(124, 232)$. This is valid as well, although none of these two lines actually starts at the specified point, and the second result even contains a small gap of some pixels. If there is a query for lines going down, with no points specified, the three vertical lines will be returned; querying for lines going up (again without specifying points) returns the same three lines, but this time the coordinates of start point and end point are interchanged in the result. As final example, a query for a line going down right

from $(150, 20)$ to $(425, 25)$ yields no result. Although there is a line connecting these two points (once again the line model allows for some imprecision), this line is horizontal, and does not go down right.

Querying the arc model is very similar. A query for an arc in quadrant 1, clockwise, starting from point $(151, 63)$, yields one result, namely the arc to $(188, 127)$. The same query with a counter-clockwise orientation yields no result (the arc from $(128, 64)$ to $(71, 107)$ is not in quadrant 1, but 4, hence it is no valid result). For links, no example is shown. They work as lines, except for the direction.

Finally, text is queried in terms of a location. The location is described by a point, polyline, polygon, or a combination thereof (see Appendix A for examples, e.g., Appendixes A.2.2 through A.2.4 for the shapes of NSD). It is intersected with the bounding box of each text. If the result is not empty, the text is returned, otherwise it is not. Note that this concept is equivalent to attachment areas, described in Section 7.1. If no location is specified, each text may be returned, regardless of where it is written on the canvas.

5.4 Recognition of Shapes

The assembler performs the actual recognition of shapes (cf. Figure 5.10). It both accesses the models and the search plan that has been computed from the specification. The assembler yields the shapes that can be identified from these data. Determined by the search plan, the assembler queries the models for primitives, as has been described in the previous sections. The assembler then combines each result from every model with the current partial finding, and continues the search with each combination independently. The details are explained below. As mentioned in the introduction to this chapter, the assembler treats every model equally, and has no specific information about the primitives contained in a model. Accordingly, each model is queried for each primitive. For example, each model is queried for straight lines, although only the line model will return results.

Note that having different independent models is just a matter of representation; conceptually it is equivalent to have only one model that is the union of the independent models, i.e., it contains all primitives found by each transformer.

The search process is based on *states*, which are created and managed by the assembler. Each state s corresponds to a node in the search plan, denoted by $s.node$. Accordingly, each state contains a (partial) shape from the sketch that corresponds to the (partial) shapes in its search plan node. The set of shapes that are completely identified in a state s are denoted by $s.complete$. If s represents only a partial state, $s.complete$ is empty.

Algorithm 5 describes the search process performed by the assembler in detail. It yields the set Sh of all shapes that can be recognized from the models (the

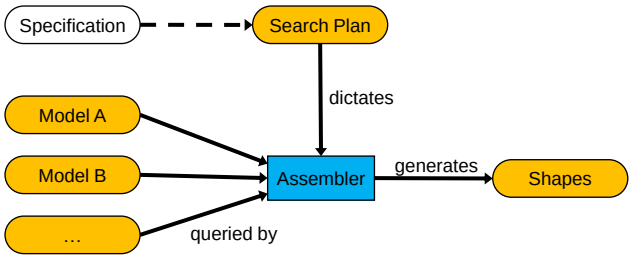


Figure 5.10: Conceptual overview of the assembler.

rounded box labeled *Shapes* in Figure 5.10).

States are processed independently; there is no connection between them. When processing a state s, two things happen: (i) all shapes that are completely identified (according to $s.node$) are added to the result (line 7), and (ii) for each child of $s.node$ in the search plan the corresponding primitives are searched for in the models (line 10). The queries to the models are as described in the previous section. For each primitive p' returned from a query, a new state is created as a copy of s with p' as additional primitive (line 12). It is checked whether this new state violates any hard constraints (line 13). If so, it is discarded; otherwise it is stored for later processing (line 14).

The assembler searches for the shapes from scratch each time (an incremental approach exhibits some challenges, which are briefly discussed in Section 5.6). It starts with exactly one empty state s that corresponds to the root of the search plan (line 3). This is the first state that is processed.

For a (partial) finding, represented by a state, the search process can be terminated for three reasons: (i) the corresponding node in the search plan has no children, as it is a leaf of the search plan, (ii) no single primitive for the transition to a child node has been found (in this case P' is empty, cf. line 10), and (iii) as already mentioned, a (partial) finding may violate hard constraints (line 13). Under certain conditions, constraints can be evaluated even before a shape is completely recognized. All constraints are based on points. The assembler can therefore evaluate a constraint, i.e., decide whether a (partial) finding satisfies the constraint, as soon as all necessary points are identified. As an example for NSD, the width of a statement can be required to be greater than its height. This can be evaluated at node `D` in Figure 5.7, as both height and width are already known at this node, and as this node represents a partial finding only for statements, but not for any other kind of shape.

As an example consider the line model shown in Figure 5.9 and the search plan shown in Figure 5.7. If we neglect text, a statement can be identified in the model, but no other shape. Consequently, in this example we only follow the leftmost trail in the search plan (to node `D`), leading to the statement without text.

Algorithm 5 Recognition of all shapes from scratch, as performed by the assembler. Recognition is based on the search plan and the models. Return value is a set of shapes Sh.

```
1: function RECOGNIZE
2:     Sh ← ∅                                          ▷ Sh will contain the result
3:     s ←CREATEEMPTYSTATE                             ▷ s is a state
4:     St ← {s}          ▷ St is the set of states that still have to be processed
5:     while St ≠ ∅ do
6:         s ←REMOVEARBITRARYSTATEFROM(St)
7:         Sh ← Sh ∪ s.complete
8:         for all (p, n) ∈GETALLCHILDREN(s.node) do
9:                                                     ▷ p is a primitive, n is a node
10:            P′ ←QUERYALLMODELSFOR(p)
11:            for all p′ ∈ P′ do
12:                s′ ←CREATESTATE(s, p′, n)
13:                if NOHARDCONSTRAINTSVIOLATED(s′) then
14:                    St ← St ∪ {s′}
15:                end if
16:            end for
17:        end for
18:    end while
19:    return Sh
20: end function
```

The transition in the search plan from the root to node `A` is triggered by a horizontal line. Querying this line leads to four results in the line model, indicated by the four figures in Figure 5.11(a) through (d). The next transition in the search plan demands a vertical line going down, starting from the left end point of the first horizontal line (indicated by `a` in the search plan). In (b), no such line is in the model, hence the result of the query is empty, and the partial finding is immediately discarded. The same holds for (c). In (a), such a line can be found, which ends in $(411, 233)$. The next transition then demands a horizontal line going right, starting at this point. No such line can be found, so this partial finding is discarded as well. What remains is (d). Here, the first vertical line can be found. It ends in point $(112, 221)$. The second vertical line, going right from this point, leads to two alternatives. The first is $(269, 232)$, which must be discarded in the next step, as there is no vertical line connecting this point to $(422, 25)$. The other alternative is $(411, 233)$. In the last step, the fourth line can be identified as well. Then the statement is fully identified (apart from text, which would now be queried by the assembler), and added to the result of the assembler. The transitions

to nodes F and L cannot be found in the line model.

Finally, it is necessary that lines are split at intersections with each other. This fact has already been mentioned in Section 4.2, but not explained so far. Again following the search plan in Figure 5.7, in Figure 5.12 the statement cannot be identified in (a) if point i is not present in the corresponding line model (in this example, letters denote points, instead of coordinates as before). The reason is that the line from c to d is identified as the only choice for a line starting in c and going right. However, from point d no vertical line connecting this point to b can be found, so the statement is not recognized. By intersecting c-d and b-e, the line from c to i is a valid alternative, and finally, since i-b is indeed vertical, the statement is identified.

For (b) the situation is similar. If point i is not present, again line c-d is the only option, and there still is no line going up, starting from point d. As a solution, the projection of point e on line c-d must be computed, and c-d must be split at this point in order to obtain two lines instead, namely c-i and i-d. Starting in point i the vertical line e-b can then be found, which completes the statement.

Runtime complexity

Let p be the number of all primitives in all models, let x be the number of primitives of the shape s with most primitives, and let c be the number of all constraints used for shapes. Note that all three variables p, x and c are independent of the proposed recognition algorithm; the value of p depends on the actual sketch, and the values of x and c depend on the specification. Then the worst-case runtime complexity of the search process can be roughly estimated by $\mathcal{O}(p^x \cdot (c+1))$, because for each of the x primitives of the shape specification there are up to p primitives from the sketch, and in each case the constraints are checked; all constraints currently supported (cf. Section 5.1) can be computed in constant time. All other shapes in the specification have equal or fewer primitives than s has, thus their complexity is equal or less. Note that $(c+1)$ is used in the estimation, and not c, since there may be no constraints specified.

Although this estimation is polynomial (c and x are fixed for a given language), in practice the runtime is roughly linear (cf. Section 10.1). This is due to several reasons:

- In practice, the number of primitives matching a query is much smaller than p. For example, if a horizontal line is queried, lines with another direction, arcs, links and text are not considered.

- Given a shape with n primitives, there are $n!$ possible orderings to assemble the shape. The search plan allows only one of them, independent of n,

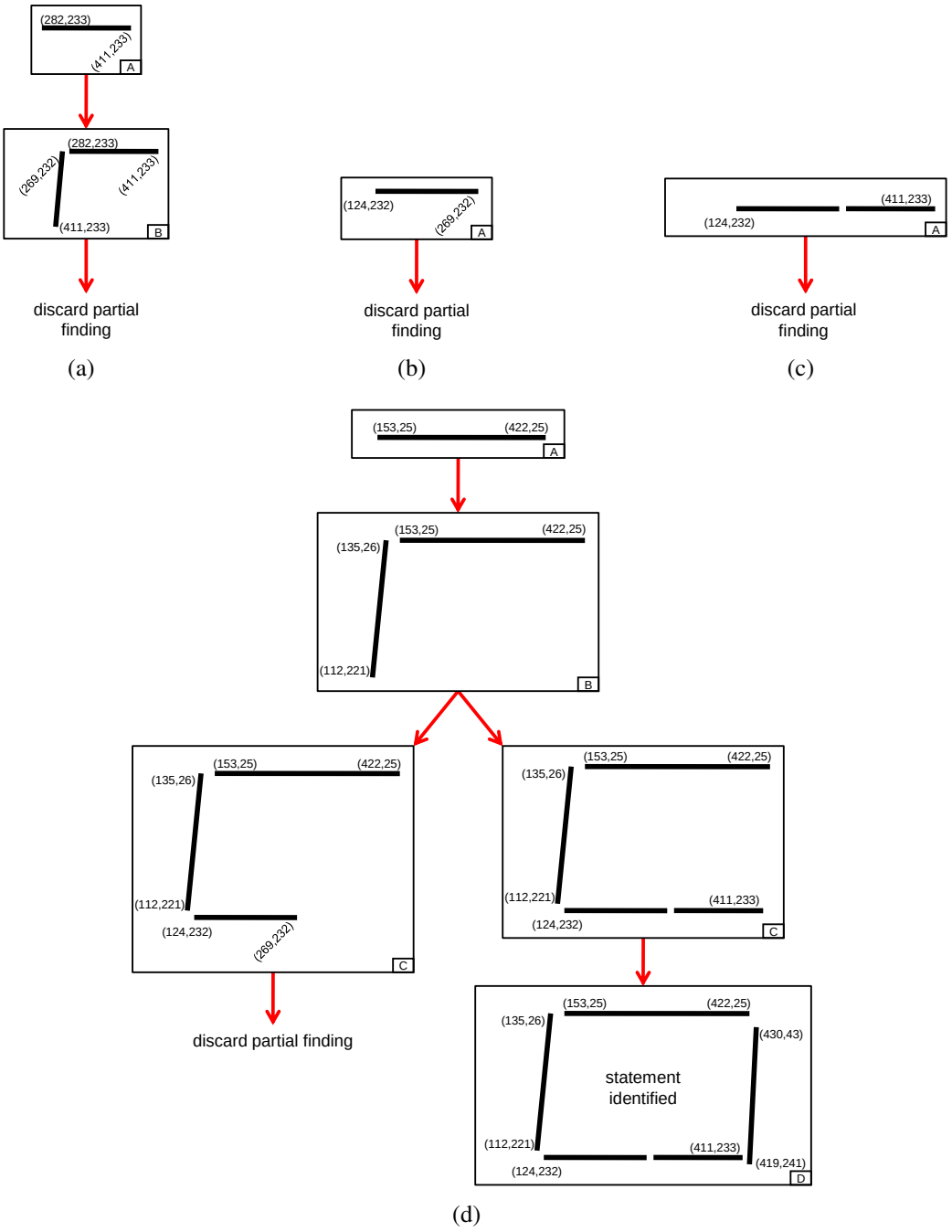


Figure 5.11: Extract from the search process for a statement in a drawing. (a)–(d) each represent a different first choice for the horizontal line. Only (d) succeeds in finding a complete shape. Partial results are discarded as indicated. The letters in the lower right corner of each state represent the corresponding search plan node in Figure 5.7. Text is neglected.

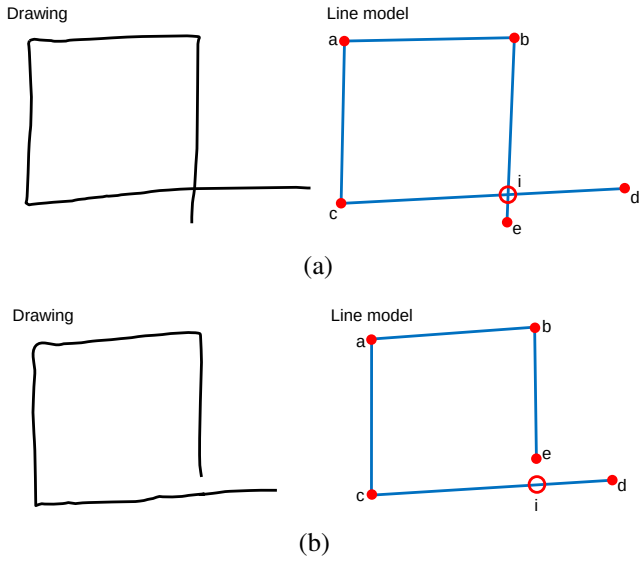


Figure 5.12: Two drawings and their corresponding line models. The NSD statement cannot be identified without intersecting lines c-d and b-e in (a), or computing the projection of point e on line c-d in (b).

which reduces runtime considerably. However, there still may be partial results that eventually turn out as dead ends, of course.

- This single ordering guarantees that the primitives are connected, which greatly reduces the number of possible primitives.
- Constraints are checked as soon as possible, pruning search plan states that cannot be completed to legal shapes.
- Primitives shared by different shapes are searched for jointly, which again means a reduction of runtime.

5.5 Assigning Ratings to Shapes

In the later steps of processing there is sometimes the need to make a choice between two shapes, or between two sets of shapes. The decision for one or the other is based on *ratings*. A rating of a shape is a positive real number, where a greater number is better. This way, that one of the two is chosen which exhibits the greater rating. Sets of shapes are rated by accumulating the individual ratings of the contained shapes. For choosing one of two shapes, or sets of shapes, it may happen that both ratings are equal. In this case, the decision for one of the two is performed in a non-deterministic way.

Computation of ratings is driven by two key ideas. (i) the more complex the shape is the higher is its rating. A shape is more complex the more primitives and constraints it is made of. The rationale behind this is that the assembler has to identify more primitives for a more complex shape, and that these primitives must satisfy more constraints, which makes the shape more valuable. (ii) the rating of a shape is higher the more precisely it is drawn. Here, the idea is to reward precise drawing style with higher ratings, and to penalize sloppy drawn shapes.

These two requirements have several consequences. One consequence is that two shapes of the same kind, e.g., two arrows, must gain the same rating, given that they are drawn with equal precision, because they are made of the same number of primitives and constraints. If two arrows are not drawn with the same precision, that one gains the higher rating that is drawn more precisely. Another consequence is that ratings are not normalized to some fixed value, as this would violate (i).

We set the upper bound for the rating of a shape to the *complexity* of the specification of the shape. This complexity is defined as the weighted sum

$$n_p \times w_p + n_{hc} \times w_{hc} + n_{sc} \times w_{sc}$$

where n_p is the number of primitives of the shape specification (with weight w_p), n_{hc} is the number of hard constraints (with weight w_{hc}), and n_{sc} is the number of soft constraints (with weight w_{sc}). All weights are positive. Obviously, the more primitives or constraints a shape specification exhibits, the greater its complexity is. In case that a shape is drawn with perfect precision, its rating is equal to its specification complexity. Otherwise, the rating is reduced in order to factor in the lack of precision.

For the three weights we suggest to set $w_p = 1.5$, $w_{hc} = 1.0$ and $w_{sc} = 0.75$. This favors primitives over constraints and hard constraints over soft constraints. Given these weights, consider the shapes defined in Appendix A.1 as examples. The arrow has a complexity of 14.25, because it consists of four primitives (4×1.5), six hard constraints (6×1.0), and three soft constraints ($3 \times .75$). Likewise, the complexity of the place is 9.0, the complexity of the transition is 7.5, and the complexity of the token is 6.0.

As mentioned above, the rating of a shape is equal to its complexity only if it is perfectly drawn and satisfies every constraint. However, an imprecise drawing style is inevitable in hand-drawing, and the assembler must allow for thresholds to also recognize shapes not drawn with perfect precision. Accordingly, the actual rating of such shapes is decreased, as the imprecise drawing style is penalized. This is done in the following way. The actual rating of a shape is computed as sum of all individual ratings for all primitives and all constraints. These are each rated by their weight, if perfectly satisfied; otherwise their rating is reduced. For

primitives, text and links are always rated with w_p, optional text that is not present with 0. The rating of a line is decreased the more the direction of the line differs from the specified direction, which can always be given by mapping the possible direction to precise values (e.g., right→0°, right up→45°, . . .). For arcs, the rating is decreased if the angle spanned by the arc is less than 90°, which may happen due to imprecise drawing. For constraints a similar approach is taken. The rating of a constraint is computed depending on the kind of comparison specified for the constraint. The more two values differ, which are constrained to be equal, the more the respective rating (given by w_{hc} or w_{sc}) is decreased. If two values are constrained to be not equal, or one greater than the other (less than, resp.), the rating of the constraint is reduced the more the two values are equal. Soft constraints, which are not satisfied, are rated by 0. Examples for practical ratings are given in Appendix C.2.

Even if a constraint is poorly satisfied, it is satisfied anyway. Accordingly, we suggest to compute the actual rating for a constraint c requiring two values to be equal by the following formula (d is the absolute difference between the two values, th is the maximum allowed absolute difference between the two values such that the constraint is still satisfied, and w is the weight, either w_{hc} or w_{sc})

$$rating(c) = (1 - \frac{d}{2 \cdot th}) \times w$$

Now even if the constraint is poorly satisfied, the constraint is still rated by 0.5 of its weight. If it is perfectly satisfied, it is rated by its weight. No higher or lower values are possible. If the constraint c requires two values to be not equal, or one greater than or less than the other, we suggest the following formula (d and w are as before, while th is the maximum allowed absolute difference between the two values to still count them as equal)

$$rating(c) = \begin{cases} w & \text{if } d \geq th, \\ (0.5 + \frac{d}{2 \cdot th}) \times w & \text{otherwise} \end{cases}$$

Using this formula, a constraint is rated by its weight if the actual distance is greater than or equal to the threshold; otherwise the rating is linearly decreased to 0.5 of its weight.

5.6 Future Work

A worthwhile extension to the approach presented in this chapter would be to have the recognition process work incrementally. Then, either after a new stroke is drawn, or when explicitly invoked by user interaction, recognition does not

have to start from scratch each time, but can reuse previously identified shapes. However, many challenges are involved in this approach. First, each transformer and model also must work incrementally, and must be able to distinguish between old data that has been used for recognition before, and new data, which has not been used so far. Second, the assembler must either store all partial findings, which is very costly, as there are many in general, or the assembler must apply some other approach to complete partial findings which could not be completed before due to missing primitives. Third, data can also be deleted from the drawing, and thus from the models, which must also be coped with. These considerations show that the incremental recognition approach outlined above requires some non-trivial bookkeeping.

As alternative, or in addition to this incremental approach, recognition could be parallelized. All partial findings are represented by search states, which are completely independent from each other. Informal testing with the sketches taken from the user study (cf. Chapter 10) has shown that the total number of search states for one run of the assembler is typically in the thousands, while the average number of search states waiting for processing is in the hundreds. Accordingly, processing these states in parallel on a multi-processor machine, which is very common hardware nowadays, could improve runtime performance significantly.

5.7 Summary

In this chapter the recognition process has been discussed in detail. The assembler queries models for primitives based on the preprocessing illustrated in Chapter 4. The order of the queries is dictated by the search plan, which is automatically computed from the specification. By using the search plan, and by evaluating constraints as soon as all necessary information is identified, states of the recognition process can be discarded as soon as possible, and unnecessary processing is avoided. The assembler does not distinguish between different models, but asks every model for a primitive. New transformers and models can be easily integrated. Each shape is rated based on its complexity, and on how precisely it is drawn. Shapes do not have to be connected. If they are not, using constraints it can be specified how the single parts are related to each other.

An interesting issue is the relationship between DSKETCH and SkG (cf. Section 3.3). The recognizer in SkG works incrementally and must thus be able to continue partially recognized shapes non-deterministically according to what the user draws. The order in which primitives are added to partial shapes is not fixed. Technically, this is done using an evolution of extended positional grammars (XPG) and a respective parser. In contrast, in DSKETCH we recognize all shapes from scratch. This is done in a deterministic manner; determinism is achieved by

the search plan. However, it is possible to map our approach to that of positional grammars [27]. In this case, primitives are taken as terminal symbols, while partially identified shapes are nonterminal symbols. This way, a search plan can be directly transformed into a suitable non-linear grammar.

Chapter 6

Postprocessing

The result from the assembler is a set of shapes, which are completely unrelated as yet. The analysis stage, which is the topic of Chapters 7 through 9, requires exactly such a set of shapes. Regardless of this, two reasons make a postprocessing step right after the assembler (cf. Figure 1.4) valuable. First, the number of shapes can be reduced without losing crucial information, as explained in Section 6.1. The less shapes that are passed to the analysis stage, the less time is consumed. The second reason for postprocessing is to deduce some extra information which can later be used to decide which shapes to add to the final result of the analysis, and which to discard. This is subject of Section 6.2. Finally, based on the diagram languages described in Chapter 2, a further configuration option is identified, which allows for some optimization of the processing. Its use depends on the diagram language, which is discussed in Section 6.3. Note that this chapter describes the single postprocessing steps in the order in which they are applied, so the removal of duplicates is first, the deduction of extra information is second, and the execution of the configuration option is third. The chapter is summarized in Section 6.4.

6.1 Elimination of Duplicates

A typical result of the recognition approach explained in the previous chapter are *duplicates*, i.e., two or more recognized shapes which represent the same actual shape drawn and intended by the user. Duplicates occur when the same drawn shape is identified more than once, each time with slightly different junction points. False positives occur among the junction points mostly due to the line transformer (Section 4.2). This transformer tends to split the input strokes more often than necessary in order not to miss an important point. An example is shown in Figure 6.1(a) and (b). Instead of four straight lines which were intended by the

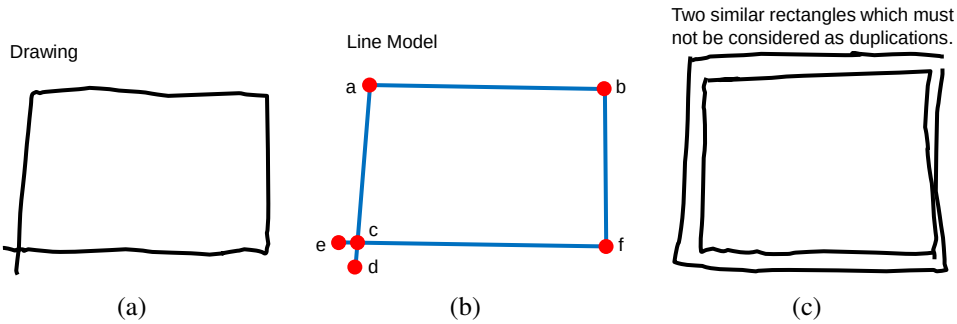


Figure 6.1: (a) a drawing of a rectangle made with one stroke. (b) the corresponding line model. (c) two rectangles which are no duplicates.

user, the assembler will get more results from the line model. Assume the search plan from Figure 5.7. In the first step, the horizontal line from a to b is returned by the line model. The second line, going down from a, leads to two different results, either c, or d. For c there are two options for the third line going right, either starting in c, or starting in e due to the thresholds allowed for answering queries. Both of them go to f. For d there are the same two options due to the thresholds. In any case, the horizontal line goes to f. From f there is then only one vertical line going to b, and the shape is fully recognized. However, four alternatives for the lower left corner of the rectangle were identified: c only, (c,e), (d,c), and (d,e). This results in four shapes identified by the assembler, three of which are duplicates.

There is another reason for duplicates. Different models may return results for the same query. It is highly unlikely that these results are equal, even if they are based on the same strokes. As an example, consider the arc model and the circle model, and a circle drawn in one stroke. The circle is identified by requesting the four arcs from the models. Both the arc model and the circle model will return valid results, but the end points of the arcs will be slightly different in general, which results in many different circles that are identified.

Duplicates impose an unnecessary load on the analysis stage, although this stage can deal with duplicates. This load results in a waste of computing resources. The challenge in the identification of duplicates is to distinguish these from two shapes which just happen to be very close to each other, as none of those must be removed. The first precondition for a shape to be a duplicate of another is that both shapes have the same type, e.g., both shapes are places, or both shapes are arrows, in the case of Petri nets. The second precondition is that both shapes comprise exactly the same text primitives if they comprise text primitives at all, e.g., two NSD statements can only be duplicates if they contain the same text. If either of the two preconditions is not satisfied, none of the two shapes is considered to be a

duplicate of the other.

To decide whether two shapes are duplicates, three conditions are checked. If one is found to be satisfied, we assume one of the two shapes to be a duplicate of the other. Then, that shape is discarded which exhibits the lower rating, because this shape is assumed to be drawn with less precision (cf. Section 5.5). Although comparing each two shapes exhibits $\mathcal{O}(n^2)$ complexity, where n is the number of shapes, the gain in terms of overall processing speed certainly warrants this expenditure, as tests made with the prototypical implementation clearly show (cf. Section 10.3).

The first of the three conditions regards the distance of the points of the two shapes to each other. The specification assigns each point of a shape a unique identifier, whether it is a junction point or a simple point. If the points from both shapes with the same identifier are *all* closer to each other than a certain, very small threshold (like 10 pixel, or less), the condition is satisfied. This condition may accidentally hold if two different shapes are drawn very close to each other (cf. Figure 6.1(c)). If this frequently happens, the threshold must be decreased.

The second condition refers to the strokes or sub-strokes used to draw a shape. This condition holds if exactly the same strokes or sub-strokes are used for both shapes, no matter how these are related to the single primitives. Obviously, it is necessary to know those strokes or sub-strokes used to draw a shape to decide about this condition. This information is preserved by the transformers, and reflected in the models. The postprocessing step can subsequently access this information and check whether the condition holds.

The third and final condition makes use of *fully connected primitives*. A fully connected primitive is a straight line, an arc, or a link where both end points are junction points, i.e., both end points are connected to other primitives. From the two examples of shapes shown in Figure 5.3, the arrow has no fully connected primitives, as each line has only one junction point. For the grid, only the four inner lines (i1-i2, i2-i3, i3-i4, and i4-i1) are fully connected primitives. Now if the strokes or sub-strokes used to draw the fully connected primitives of one of the two shapes are a subset of the strokes or sub-strokes used to draw the fully connected primitives of the other shape, the first shape is regarded as duplicate, and is removed, no matter which shape has the higher rating. The idea behind this condition is that fully connected primitives are more significant for a shape, usually describing the location and extent of the shape better, because *both* end points are not depending on one primitive only, but several primitives. Accordingly, there is less choice in the identification of fully connected primitives.

Evaluating these three conditions revealed that the first and the third already remove most of the duplicates, even if applied solely. However, in some cases it is the second condition, or a combination of all three conditions that remove more duplicates, so it is valuable to always apply all of them.

The removal of duplicates does not harm the later analysis stage in terms of losing valuable information. As one shape is always kept, and only its duplicates are removed, the essential data is preserved.

6.2 Identification of Conflicts

Depending on the diagram language, one stroke or sub-stroke may only contribute to one shape at the same time, and not to two or more. Consequently, whenever the same stroke or sub-stroke is used for different shapes, only one of the shapes can be correct, and the others are false positives. This is true no matter whether the shapes in question have the same specification, or not.

Many diagram languages exhibit the described behavior. From the six examples given in Chapter 2, it is true for all but Nassi-Shneiderman diagrams. For NSDs, almost each stroke or sub-stroke is shared between two shapes, except for those strokes or sub-strokes forming the very outline of a diagram, and for the diagonal lines used in conditions. In the other five diagram languages each stroke or sub-stroke may only belong to one shape.

If two shapes are found to use the same stroke or sub-stroke this is called a *conflict*. The final result of processing a sketch has to be free of conflicting shapes. Conflicts cannot be solved in the postprocessing step, but they can be detected in this step. A solution is not possible, because no context information is available as yet. The solution could only be based on meta-rules, which is avoided as much as possible in DSKETCH. The subsequent analysis stage establishes context of shapes and thus has the necessary means to solve the conflicts.

As mentioned, the existence of conflicts depends on the diagram language. Hence, the specification of a diagram language also has to define if conflicts should be detected, or not. If yes, the result can be expressed as a binary symmetric relation between shapes, where each tuple in the relation refers to two conflicting shapes.

6.3 Suppression of Shapes Containing Other Shapes

Next to the option introduced in the previous section regarding conflicts between shapes, a further configuration option, very specific to NSD, is reasonable. The graphical appearance of all shapes is based on blocks, and such blocks are also drawn next to each other. Many false positives are recognized, as each two neighboring blocks can be recursively combined to a larger block. As for the duplicates, the analysis stage can deal with this circumstance, but again performance suffers severely. The NSD shown in Figure 6.2(c) consists of nine shapes: one

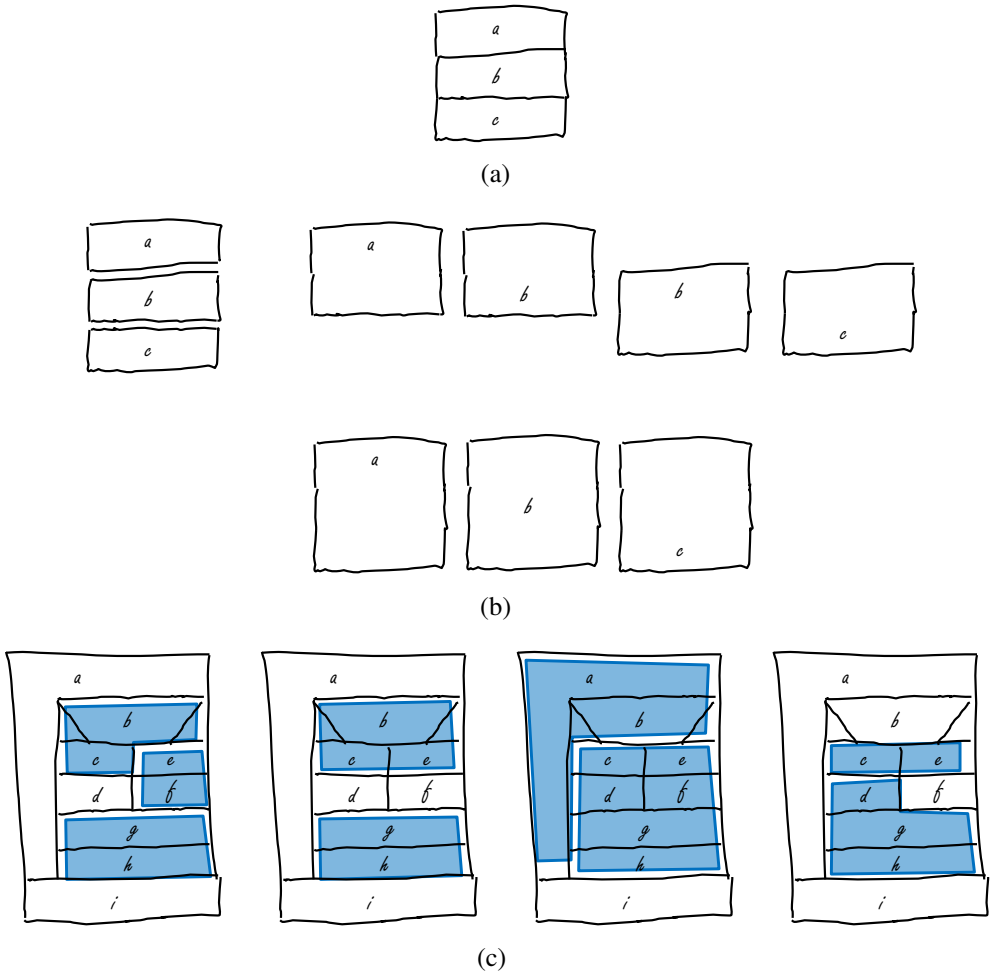


Figure 6.2: NSDs and examples of false positives. (a) a simple NSD comprised of three consecutive statements. (b) all ten statements recognized in the simple NSD, seven of which are false positives. (c) further examples of false positives for a more complex NSD.

while-loop, one condition, and seven statements. The prototypical implementation identified 41 shapes[1]. Some of the false positives which are identified due to the special nature of NSD are also shown in the figure.

To make this point more clear, consider two consecutive statements, like g-h in the figure. As the examples show, instead of the two statements four are identified (the highlighted larger statement can be combined with text g or h, which makes two more solutions). The problem gets worse the more consecutive statements

[1]The implementation counts each shape only one time, even if there are different texts that can be combined with the shape. Otherwise, there would be identified by far more than 41 shapes.

there are. For three consecutive statements (cf. Figure 6.2(a)), the assembler identifies ten statements (cf. (b)), for four consecutive statements 20 are identified. In general, for n consecutive statements $\mathcal{O}(n^2)$ statements are identified.

Again the solution is based on a meta-rule. The option allows for suppression of those shapes which completely contain another shape, no matter what the shapes are. This way, only the smallest shapes are preserved, which is exactly the right thing to do here, as all false positives combine two or more smaller shapes.

6.4 Summary

The need for a postprocessing step is shown in this chapter. Its three goals are to remove duplicates, identify conflicts, and suppress shapes containing other shapes. Removal of duplicates minimizes load for the analysis stage, conflicts can be solved by the analysis stage, but can be identified during postprocessing, and suppressing of shapes containing other shapes allows for tuning the approach for the characteristics of diagram languages like NSD.

Each of the options is deduced from observations of diagram languages and how DSKETCH works with these languages. One of the options controls the identification of conflicts, while the other is based on a meta-rule, with the goal of minimizing the load for analysis. Although a fair selection of diagram languages has been examined (cf. Chapter 2), it cannot be excluded that further diagram languages impose a need for further options.

Chapter 7

Modeler

An overview of the analysis stage is shown in Figure 1.4. The modeler, described in this chapter, is the first of three steps in the analysis stage of processing a sketch. It creates a graph model from the shapes identified in the recognition stage. The two steps following the modeler are the reducer and the parser, which are explained in the next two chapters.

The complete analysis stage is based on DIAGEN [69, 68]. DIAGEN is a tool to generate editors for visual languages from specifications. In fact, this also describes DSKETCH, with the only exception that the editors generated by DIAGEN do not support sketching. Also, as mentioned in Section 1.2, the generated editors provide support for free-hand editing, which is necessary for the combination with sketching. Accordingly, DIAGEN makes the perfect start for DSKETCH. Editors generated by DIAGEN exhibit the architecture shown in Figure 7.1. Data structures are shown as rectangles, processing units are shown as rounded boxes with underlying shadow, and arrows denote flow of control. The user is provided with a GUI, the *drawing tool*, where he can create a diagram in a traditional point-and-click fashion. Processing the diagram is then split into four steps: *modeler*, *reducer*, *parser*, and *attribute evaluation*. The modeler first creates a hypergraph model representing all *components* the diagram consists of (what we refer to as *shapes*), and the spatial relations between the components. Result is the *hypergraph model*, which is then reduced. Goal of this step is to decrease the size of the model, and to discard some syntactically invalid patterns. This step yields the *reduced hypergraph model*. The hypergraph parser creates a derivation structure by applying a bottom-up parser, taking the edges in the reduced model for terminal symbols. Finally, attributes of the derivation structure can be evaluated in order to produce the *semantic representation* as final result. These steps, beginning with the modeler, are completely adopted in DSKETCH. This chapter explains the modeler in detail, and the modifications that were necessary in order to fulfill the needs of DSKETCH. The next two chapters do so for the reducer and the parser (attribute

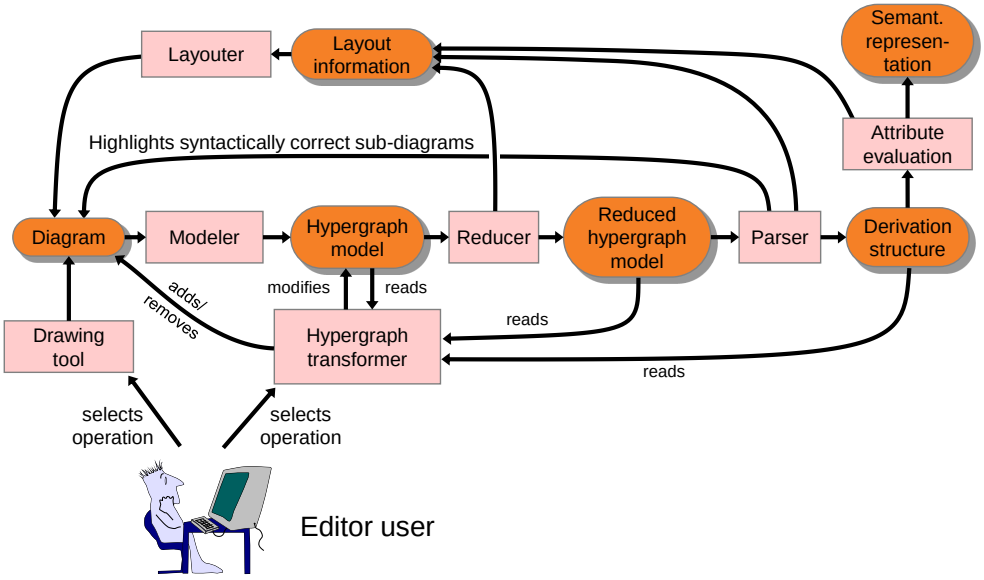


Figure 7.1: Architecture of an editor generated by DIAGEN.

evaluation is part of the chapter on the parser). Aspects of DIAGEN not relevant to DSKETCH are the *layouter* that can rearrange the components of a diagram to gain a visually more appealing diagram, and the *hypergraph transformer*, which allows for structured editing of diagrams using predefined editing operations.

Applied to our topic, the fundamental task of the modeler is to relate the shapes identified in the recognition stage, according to some rules, and create the *hypergraph model* (HM) which represents both the shapes and the relations. A graph structure is required because the reducer and the parser essentially apply graph rewriting techniques. The rules for relating shapes are given by the specification of a diagram language. To specify what relations between shapes exist, it is first necessary to describe which regions of shapes can be related at all. These regions are called *attachment areas*, described in Section 7.1. Thereafter, relations between the attachment areas may be defined, which is explained in Section 7.2. Before the actual creation of the hypergraph model is illustrated in Section 7.4, it is first necessary to introduce hypergraphs (Section 7.3), which are the central data structure for the analysis stage. The chapter is summarized in Section 7.5.

7.1 Attachment Areas

An *attachment area* describes what regions of a shape can be related to other shapes (a shape is never related to itself). Each shape should have one attachment area at least; otherwise it cannot be related to other shapes. The definition of

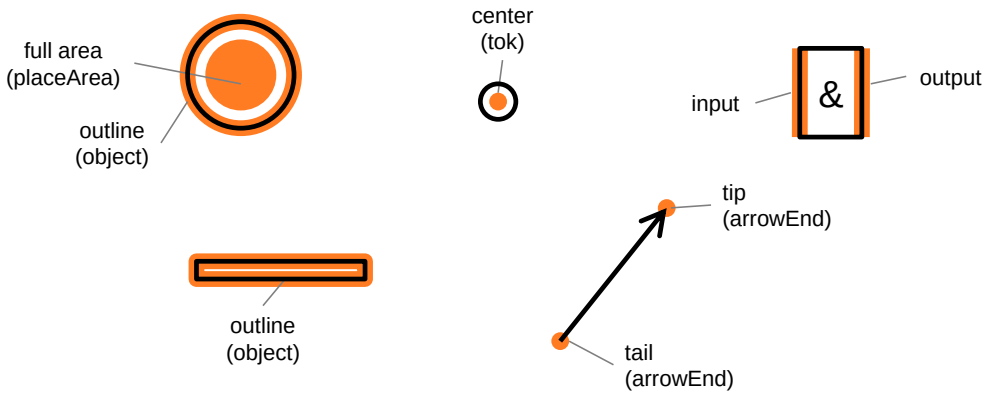


Figure 7.2: Examples of shapes and their attachment areas. Labels are given in parentheses.

attachment areas is part of the specification of a shape. Examples are shown in Figure 7.2. An arrow, for instance, usually has two attachment areas, which are its tip, where the arrow is attached to its sink, and the point where it is attached to its source. In a Petri net, both the place and the transition have their outline as attachment areas, because arrows start and end at the outline of places and transitions. The place may have its full area as another attachment area, because tokens can be placed anywhere inside places. Tokens, on the other hand, may also have their full area as attachment area. Alternatively, only the center point could be specified as attachment area. The operator shape from BLDs has two attachment areas, too, which are its left vertical line for input, and its right vertical line for output.

These examples show that an attachment area can either be a point (arrow, token), a polyline which may or may not be closed (place, transition, operator), or a polygon (place, token). In fact, a single attachment area can even be an arbitrary combination of these three basic building blocks. For example, it is possible to specify one attachment area for stick figures (cf. Figure 1.5) composed of the area of the head, both legs, and the outer end points of the lines forming the arms. Attachment areas are specified in terms of junction points, simple points and primitives. Examples can be seen in Appendix A.

Unlike DIAGEN, due to the impreciseness of hand-drawing, actual attachment areas of shapes may be heavily deformed. As an example consider again the transition from a Petri net. Its only attachment area is a rectangle parallel to the axes. Figure 7.3 shows the drawing of a transition, along with this rectangle (shown in a dashed style). It can be seen that the actual strokes differ noticeable from the perfect, precise attachment area (a). However, when the user wants to draw an arrow, say, starting from this transition to some place, he does not care about the perfect

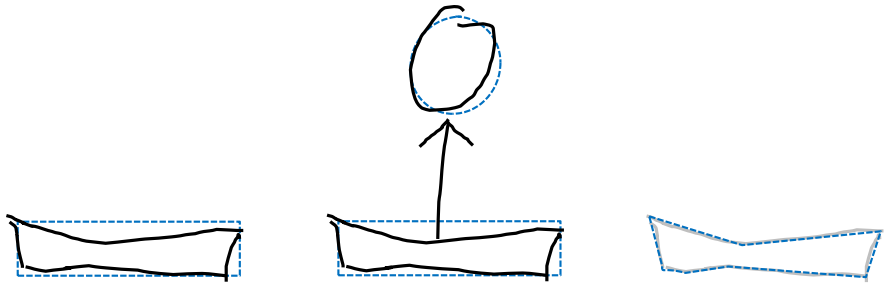


Figure 7.3: (a) a transition which is deformed. (b) an arrow respecting the deformation, along with a deformed place. (c) a better approximation to the transition using seven segments.

rectangular attachment area. Instead, he assumes his own strokes as attachment area. The same holds for the sink of the arrow, which is a place. It also shows clear deformations (b).

To account for these issues, when computing attachment areas of shapes, the actual strokes must be considered, instead of the idealized specification of shapes. Thus, the polyline for the attachment area of the transition must have more than just four segments (for the four sides of the rectangle) to follow the actual stroke more precisely. In this example, even a small increase to seven segments improves the result a lot, as (c) shows. To simplify the approximation by the polyline, the actual stroke (or strokes) are taken, which are polylines themselves. For the place, comprised of arcs, the same approach is followed. By taking the strokes instead of the actual primitives, arcs are effectively done away with. This simplifies later computation. Polygons describing attachment areas can be computed in the same way. The amount of data can even be reduced by taking only every n-th point of a stroke, where n is a small number like 3. While this means an obvious reduction of precision, in practice the sampling rate is so high that this does not matter; the result is a still very precise polyline or polygon which can be easily computed.

Points (simple or junction) and primitives are directly given by the sketch. However, in some cases they do not suffice to specify an attachment area. Figure 7.4 shows the grid from Tic-tac-toe. It comprises nine cells, each of which may contain a mark. The desired attachment areas are shown as rectangles. However, as indicated by the question marks in the figure, for each of the four corner cells a point is missing to describe the attachment area properly. Luckily, the coordinates for these four points can be computed from points given by the sketch. For example, the upper left corner's point, called ul in the following, has the same x-coordinate as the simple point l1, and the same y-coordinate as t1. By computing ul from other, given points, new straight lines can be constructed as well.

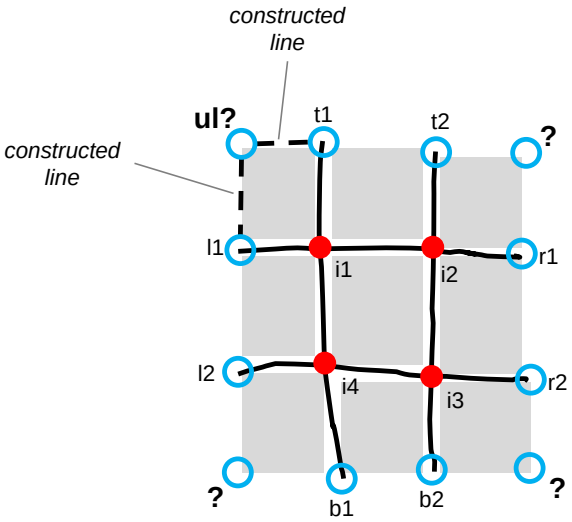


Figure 7.4: The grid from Tic-tac-toe requires computing the points indicated by question marks based on the points given by the sketch, in order to define the attachment areas in the corners (shown by the rectangles). The constructed lines (dashed lines) result from the computed points.

In this case, a straight line from l1 to ul, and another one from t1 to ul. Using these two lines and the sketched lines from l1 to i1 and from t1 to i1, the polygon describing the attachment area can be defined[1].

To conclude, attachment areas are composed of an arbitrary number of points, polylines and polygons. These are specified in terms of junction points, simple points and primitives. Sometimes it may also be necessary to compute additional points and primitives from the ones given by the sketch. Both the polylines and the polygons consist of straight segments only. Their computation is based on the original strokes drawn by the user, which are already polylines themselves. In case of a constructed line, no stroke is present, and the line is represented as one segment between the points it connects.

[1]Note that DIAGEN additionally supports the concept of infinite attachment areas, which are constrained by geometric properties, e.g. all points to the left of a given line. Using this special kind of attachment area, there is an alternative in specifying the grid cells: for each of the four lines of the grid, two infinite attachment areas can be specified for each line, one containing all points to the left of the line and one containing all points to the right of the line for the two vertical lines, or above the line and below the line for the two horizontal lines. This way, a mark can be specified to be inside a cell if it is inside the intersection of two or more of the eight attachment areas. However, DSKETCH does not support the concept of infinite attachment areas.

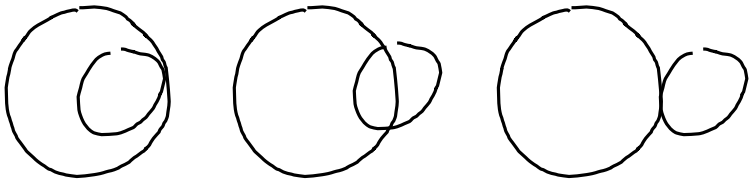


Figure 7.5: Three different cases of two related circles. The three cases cannot be distinguished by relation types only, but require further conditions.

7.2 Relations

A *relation* between two different shapes is given if two attachment areas, one from each shape, are overlapping. However, not every pair of overlapping attachment areas is relevant. Part of the specification of a visual language is to describe which attachment areas can be related at all. This is described by *relation types*. Relation types describe how shapes may be related to each other by telling what attachment areas may be related. In order to distinguish between attachment areas, each area is labeled in the specification, as shown in Figure 7.2 (the labels are written inside the parentheses). Labels may be reused for different attachment areas, or they may be unique for an attachment area. This decision is completely free to the writer of the specification.

Again Petri nets are considered as example. Both attachment areas of the arrow can be labeled with arrowEnd, for example. The outline of place and transition can be labeled with object. Then a relation type can be defined between object and arrowEnd, and by doing so arrows can be attached to places and transitions. The area of the place can be labeled placeArea, and the attachment area of the token (whether it is a polyline, area, or point) can be labeled tok. Then a relation type can be specified between placeArea and tok to indicate that tokens can be related to the area of places.

To distinguish relation types from each other, they are labeled, too. This is necessary for subsequent steps of the analysis stage. For example, the relation type between object and arrowEnd can be labeled by attachedTo, and the relation type between placeArea and tok by contains.

Relation types may have a condition. The condition allows restricting the presence of relations. For example, let a circle have its outline as attachment area, and let there be a relation type that allows for relating two circles. Then, using relations alone it cannot be distinguished for two related circles whether one circle contains the other, or the circles lines intersect, or the circles are next to each other (cf. Figure 7.5). Using conditions, these cases can be distinguished. Note that the original DIAGEN already supported conditions.

Finally, there is a subtle distinction of relation types. The arrow in Figure 7.3 is

clearly meant to connect the transition to the place, although it neither touches the place nor the transition. Obviously, some threshold has to be allowed for, because shapes will never be neatly aligned when drawn by hand. The larger the threshold is, the more forgiving the system is in terms of sloppily connected shapes. On the other hand, a token inside a place does not need any threshold. Even when drawn by hand, the token can be easily placed *inside* the place. Even more, if some threshold is applied, a token drawn *next to* a place would also be recognized as inside the place, according to the relation type contains from above. This means that for some relation types a threshold is absolutely necessary, while for others it is harmful and allows for a wrong interpretation. The later kind of relation type is called *rigid* to indicate that no threshold is considered, while the former kind of relation type is *not rigid*. Whether a relation type is meant to be rigid or not can be specified as part of the condition. In DIAGEN, no such large thresholds need to be allowed for, as the input is much more precise. Accordingly, the notion of rigid is not necessary.

7.3 Hypergraphs

As mentioned in the introduction to this chapter, both shapes and relations are represented in a *hypergraph* [9] in DIAGEN. A hypergraph is like a regular graph, but each edge may be connected to an arbitrary number of nodes. When speaking of a hypergraph, an edge is said to *visit* nodes. Each edge and node may be labeled, although in DIAGEN only edges are labeled, nodes are not. The *arity* of an edge, i.e., the number of nodes it visits, depends on the label of the edge. In this sense, a regular graph can be seen as a hypergraph where the arity of each edge is 2. In a hypergraph, edges are called *hyperedges*, and nodes are called *hypernodes*. Nevertheless, the terms *edge*, *node* and *graph* will often be used if it is clear from the context that they refer to a hypergraph. Hyperedges with an arity of 1 are called *unary*; hyperedges with an arity of 2 are called *binary*.

Hypergraphs are well suited for a graphical representation. Figure 7.6 shows a hypergraph consisting of seven edges and four nodes. Nodes are shown as filled circles. Edges are shown as boxes with the label of the edge written inside. Sometimes, as for the edge labeled c in the figure, edges are not shown as boxes, but as polygons. Each node visited by an edge is connected to that edge by a line, called a *tentacle*. To distinguish the nodes visited by an edge, the tentacles are numbered. For unary edges, this is not necessary. Edges and nodes may be given a unique *name* to distinguish them from other edges and nodes. In the case of nodes, the name is written next to the node. In the case of edges, the name is written inside the edge, separated from the label by a colon, e.g. n1:a. For binary edges there is a special notation. They can be drawn as bold arrows from the first visited node

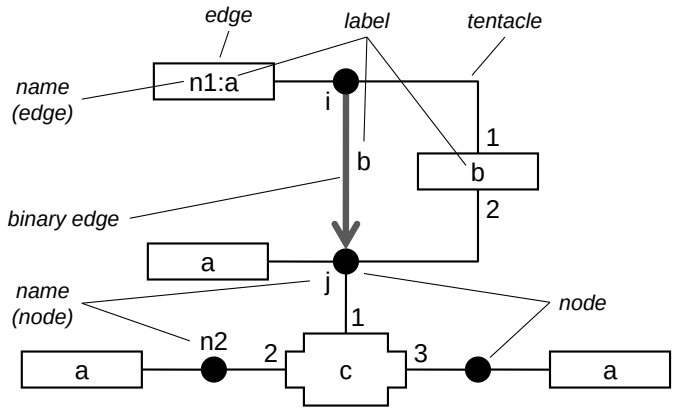


Figure 7.6: A hypergraph consisting of seven edges and four nodes.

to the second. Both edges labeled b in the figure thus are equivalent binary edges between the same two nodes i and j.

Furthermore, each hyperedge may have *attributes*. An attribute is a *(name, value)* pair. The name of an attribute must be unique for the edge; different edges may have attributes with the same name. Attributes of an edge depend on the label of the edge, i.e., different edges with the same label also have the same attributes, although with different values in general. Attributes are not shown in the graphical representation shown in the figure. For an edge named n, the attribute attr is denoted as n.attr.

7.4 Creating the Hypergraph Model

First, the modeler establishes all relations between the attachment areas of all shapes. Then, the hypergraph model (HM) is created. It represents all shapes, all relations, and all conflicts. Section 7.1 describes how attachment areas are computed based on the specification of shapes.

For each pair of attachment areas labeled a and b where a relation type r between a and b is specified, the distance between a and b is calculated (see below). Then, this distance is compared to a threshold (set to 40 in the prototypical implementation). This allows for convenient drawing of shapes, as they do not have to stick together very closely. If the distance between the attachment areas is not greater than the threshold, and if the condition of the relation type holds (if there is one), an actual relation of type r between a and b is found. In case that r is rigid, there is no threshold allowed and the two attachment areas must touch each other, i.e., their distance is 0.

Let $\{a_1, \ldots, a_i, \ldots, a_n\}$ be the constituents of attachment area a, i.e., each

$a_i, 1 \leq i \leq n$ is either a point, a polyline, or a polygon. Let $\{b_1, \ldots, b_j, \ldots, b_m\}$ be the constituents of attachment area b. Then the distance between a and b is the minimum of all distances between each a_i and each b_j, so

$$\text{dist}(a, b) := \min\{\text{dist}(a_i, b_j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Points, polylines and polygons can all be seen as (possibly infinite) sets of points. The distance between a_i and b_j then is the minimum of all Euclidean distances between each two points from these two sets, so

$$\text{dist}(a_i, b_j) := \min\{\|s - t\| \mid s \in a_i, t \in b_j\}$$

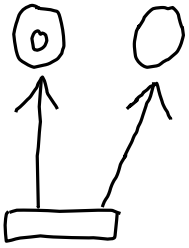
Using algorithmic geometry, $\text{dist}(a_i, b_j)$ can be computed.

For the creation of the HM, the modeler represents each shape as edge labeled with the name of the shape, e.g., place or transition in the case of Petri nets. These edges are called *shape edges*. All attachment areas of all shapes are represented by a unique node each. Each shape edge visits exactly those nodes which represent its attachment areas, and the nodes are visited in the order in which the attachment areas are specified for the shape. As a result, each node is visited by exactly one shape edge in the HM.

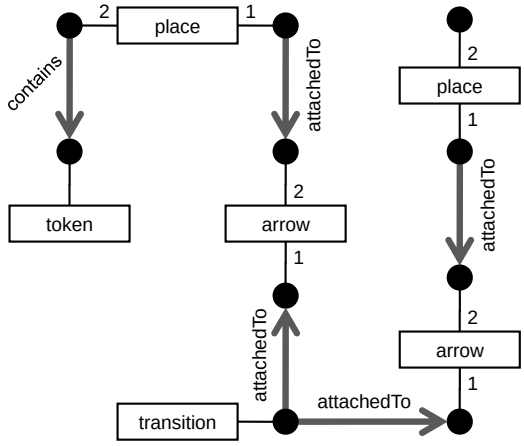
Relations between shapes are represented as binary edges, which are called *relation edges*. The label of the corresponding relation type is also used as label for the relation edge. In the graphical notation, the special notation with the bold arrows is used to represent relation edges. Relation edges always visit the two nodes representing the attachment areas which are related. The attachment area which is specified first for the relation type is visited first, too. In the example from Section 7.2 the attachedTo relation always visits the node representing the object area first and the node representing the arrowEnd area second.

A drawing of a Petri net with one transition, two places, two arrows and one token is shown in Figure 7.7(a), along with two HMs, given that there are no duplicates. The first HM (b) is the one that is actually intended. The six shapes from the drawing are represented by one shape edge each, and five relation edges attach the two arrows to their sources and targets, and the token inside the left place. Note that the direction of the arrows in the drawing can be seen in the HM by the numbering of the tentacles. The source of the arrows is always numbered with 1, the target with 2.

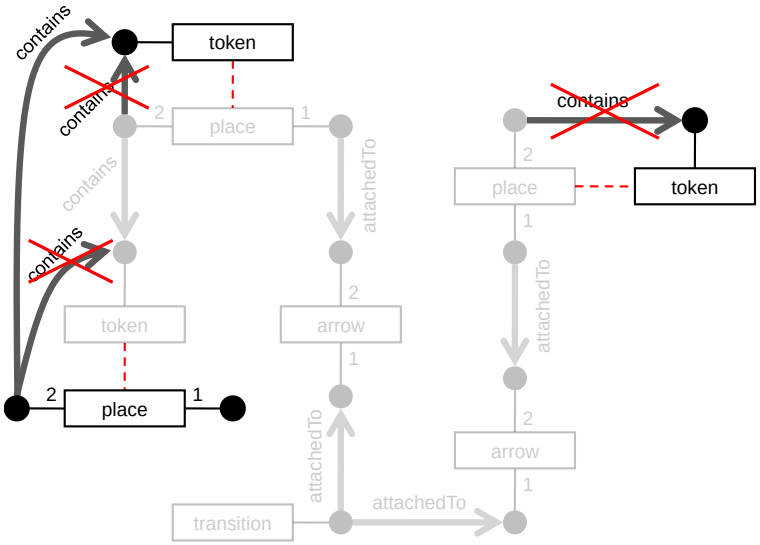
However, because a circle is defined as the shape for both a place and a token, from the three circles in the drawing three places *and* three tokens are recognized, and must be represented in the actual model (c). HM (b) is a subset of (c); it is still valid, but grayed out to put emphasis on the extra information. The dashed lines indicate the conflicts which were found in the postprocessing step, because a token and a place recognized from the same circle use the same stroke (cf. Section 6.2).



(a)



(b)



(c)

Figure 7.7: (a) hand-drawn Petri net. (b) desired HM of the Petri net. (c) actual HM; the desired HM is grayed out, conflicts are shown by dashed lines. The crossed out relation edges are correct, but suppressed due to the conflicts.

Two shape edges are called *conflicting* if the shapes they represent have a conflict. This circumstance is always shown by dashed lines.

As more shapes are found, the modeler also identifies more relations between those. What surprises initially, but is nevertheless correct, is that a place always contains the token which is recognized from the same stroke, although the respective shape edges are conflicting. While shown in the figure for the sake of clarity, the modeler actually does not regard relations between conflicting shapes, and does not create respective relation edges in the HM for those relations. Recall that a conflict between shapes means that only one of them can be correct, and one is definitely false, so relations are of no interest between conflicting shapes. All the extra information in (c) is not intended and will be dealt with in the next two steps of analysis, the reducer and the parser.

Shape edges are attributed based on the shapes they represent. For each text primitive in the shape (defined in the specification), an attribute is generated automatically, with the name of the text as name of the attribute, and the actual text as value of the attribute. The same is done for the rating of the shape, which is represented as an attribute named rating. Other attributes cannot be generated automatically, but have to be specified. They are always based on computations on the (junction) points of the shape. Take a place, for example, where center and radius can be computed and stored as attributes of the respective shape edges. These attributes can then be used by the reducer and the parser.

The modeler as described in this section differs only in details from the original DIAGEN modeler. Our application of the modeler additionally considers deformed attachment areas and conflicts between shapes, and automatically adds attributes representing the values of text primitives and the rating.

7.5 Summary

Based on attachment areas of shapes, and relation types specified between these, the modeler creates a hypergraph model called HM. This model is used by the subsequent processing steps to complete analysis. In the HM, each shape is represented as a shape edge, and each relation is represented as a binary relation edge. The identification of relations depends on the kind of attachment areas, which can be a combination of points, polylines and polygons. Relation types can have an optional condition. Using this condition it can be specified, for example, whether a relation type is rigid or not, i.e., whether a threshold may be allowed for the identification of an actual relation. Conflicts are also represented in the HM, as well as attributes, which can be attached to edges. The modeler assures that no relation between conflicting edges is represented in the HM.

Compared to the original modeler found in DIAGEN, DSKETCH makes the

following additions and changes (all other aspects are left unchanged): attachment areas may be deformed in DSKETCH, a larger threshold is applied for the identification of relations, relation types may be rigid, conflicts are considered, and some attributes and their values are computed and added automatically. This means that the notion of attachment areas, relations, relation types, conditions for relation types, and using a hypergraph representation have already been part of DIAGEN.

Chapter 8

Reducer

In the second step of the analysis stage, the HM yielded by the modeler is reduced to the *reduced hypergraph model* (RHM). The RHM is then processed by the parser as final step in the analysis stage, as shown in Figure 1.4. The parser is discussed in the next chapter.

The need for a *reducer*, explained in this chapter, stems from two considerations: (i) a model with less nodes and edges can be expected to be processed more efficiently, and (ii) there may be syntactically invalid patterns which can be easily identified and removed. Of course, by removing invalid patterns, the size of the graph decreases automatically. Furthermore the structure of the HM is generic in that each edge is either a shape edge or a relation edge, no matter what the domain is. Using specific knowledge of the domain, a more compact representation of the HM can be achieved. The structure of the RHM is no longer generic, but depends on the domain.

The reduction process is guided by *reduction rules*. These are part of the specification of a diagram language. Because the application of reduction rules is related to graph transformation, Section 8.1 gives a brief introduction into this topic first. Then, Section 8.2 explains reduction rules and their application in detail. In Section 8.3 some issues are investigated that are emerging from our application of the DIAGEN system to sketching. A quick glance at future work is given in Section 8.4. Finally, Section 8.5 summarizes the chapter.

8.1 Graph Transformation

Graph transformation means to modify a given graph G by the application of graph transformation rules. In our case, as well as in DIAGEN's, we deal with hypergraphs, which have been introduced in Section 7.3. A graph transformation rule is given by a *left-hand side* (LHS) L and a *right-hand side* (RHS) R, both of

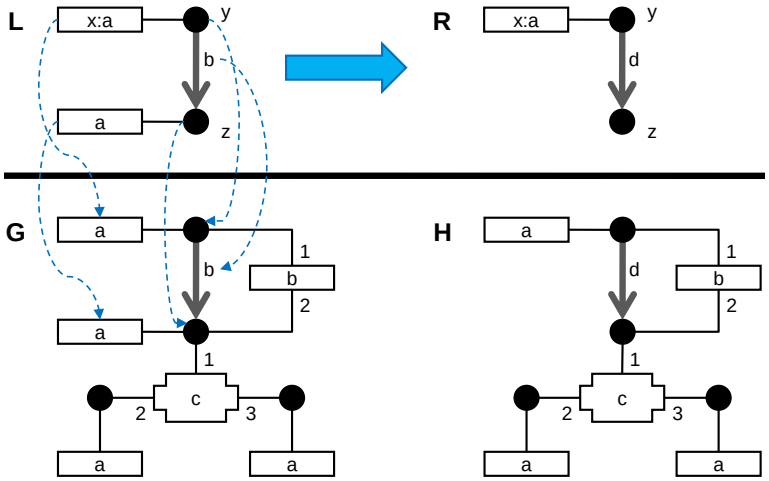


Figure 8.1: A graph transformation rule with LHS L and RHS R, and its application to a graph G, yielding a graph H.

which are graphs (hypergraphs in our case). The idea of graph transformation is to identify an occurrence of L in G (called a *match*) and replace that occurrence with R to yield the resulting graph H. One application of a rule is called *direct derivation*. A transformation of one graph into another can then be given as a list of direct derivations.

The key idea of a direct derivation is to delete those *objects* (edges and nodes) in G which are in L but not in R, to add those objects which are in R and not in L, and to preserve those objects which are both in L and R. An example is the rule given in Figure 8.1. L and R are separated by a bold arrow. Identical objects in L and R are given the same name (x, y, and z). One of two possible matches of L in the graph shown in Figure 7.6 is shown below the bold horizontal line, which separates the rule from its application to G yielding H. For the match, each object in L must be identified in G, which is achieved by finding a morphism between L and G. This is indicated by thin dashed arrows in the figure. The resulting graph H of applying the rule is also shown. It can be seen that all objects which are both in L and R are still present in H (x, y, and z), that all objects which are in R but not in L are added (the edge labeled d), and that all objects in L but not in R are removed (the unnamed edges labeled a and b). An introduction to graph transformation is given in [51]; a detailed and formal discussion is given in [82].

As mentioned above, there is a second match for L in G, where the edge labeled b is different. What can be seen in the figure is that this second match can only be found in G, but not in H. Accordingly, after the rule is applied the first time, no matter for which of the two matches for L, it cannot be applied a second time in this example. It follows that the result is influenced by the ordering in

which rules are applied.

8.2 Reduction Rules

The rule application performed by the reducer is different from the general approach to graph transformation described in the previous section. Reduction rules also have an LHS and RHS (and more, see below), and matches for the LHS are searched for in the HM, but the HM is not modified by the reducer. Instead, a different model, the reduced hypergraph model (RHM) is modified. Initially, it is empty. For each match of an LHS in the HM all objects in the corresponding RHS are added to the RHM. The effect is that the same object in the HM can be matched by different LHSs, even if it is not in the corresponding RHSs. This would not be possible with graph transformation as described in the previous section. Also, no object is ever deleted in this process. An example is shown in Figure 8.2. We assume the same rule as in Figure 8.1, with the exception that the node named z now is different in L and R. There are two matches for the LHS of the rule in the HM. The RHM shown in Figure 8.2 comprises all objects from the respective RHSs R' and R''. Although rule applications are independent of each other, edges x' and x" are equal in the RHM, because they occur both in the LHS and the RHS of the rule, and because they are matched by the same edge in the HM. A similar observation holds for nodes y' and y", so these two are also equal in the HM. The edges named d' and d", and the two unique nodes they visit, only occur in the RHSs R' and R'' of the rule applications, but not in the LHSs, so they are not equal in the RHM. See [68] for a detailed description of the semantics of the DIAGEN reducer.

The reducer applies reduction rules in the described way for each match of an LHS of a reduction rule. As the number of nodes and edges in the HM is finite, and the number of reduction rules is also finite, this process certainly terminates. Note that the parser (cf. Chapter 9) works with the RHM *only*, so any necessary information which is not transformed by a suitable rule is lost. By convention, labels used for the edges in the RHM are different from the labels used for edges in the HM.

A reduction rule consists of five parts, (i) an LHS, (ii) a RHS, (iii) an optional condition, (iv) an optional action, and (iv) a (possibly empty) set of *negative application conditions* (NACs). (i) and (ii) have been described above. If a match for an LHS is identified, the rule is applied, and the action (iv) is processed, if there is one. Actions are used to set attributes of the edges in the RHS. If there is a condition (iii), it must hold; otherwise the rule is not applied. Finally, rules may have attached one or more NACs, where each may have an own condition. A NAC is again a graph, and it restricts the application of a rule. If a match for the

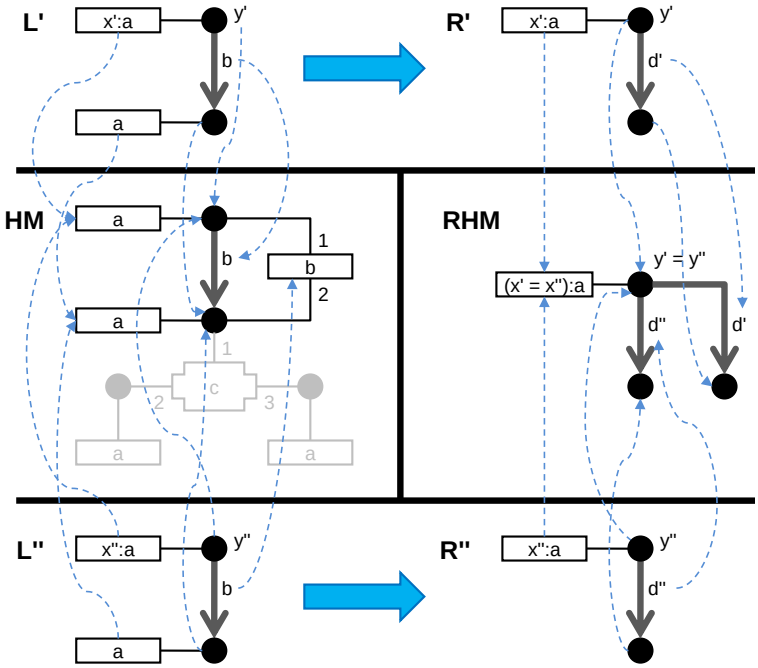


Figure 8.2: Two applications of a reduction rule similar to the one in Figure 8.1, and the resulting RHM. Those nodes and edges in the HM that do not occur in matches of the LHS are grayed out.

NAC is found in the HM, rule application is prohibited. Just like for the LHS, the condition of the NAC must hold for a successful match. Examples for NACs are given below.

We continue the example of Petri nets given in Section 7.2. First, a new attachment area transArea is specified for transitions, which is made by the full area covered by a transition. Then, a relation type touchTP from transArea to placeArea is added. Using this relation type it can later be determined if a place and a transition touch each other or overlap, which we do not allow in this example. Because overlapping or touching transitions and places are also not allowed, a second relation type touchTT from transArea to transArea is added, and a third relation type touchPP from placeArea to placeArea.

Assuming the new attachment area transArea and the three new touchXX relation types, and everything that has been specified before in Section 7.2, the reduction rules for Petri nets are shown in Figure 8.3. The bold arrows pointing from left to right separate the LHS from the RHS. By convention, all edges in the RHS of reduction rules have labels starting with t_. This convention is explained in the next chapter. Conditions are given textually (cf. 8.3(c)). They always refer to edges in the LHS only. In rule (c) we assume that both for edges labeled place

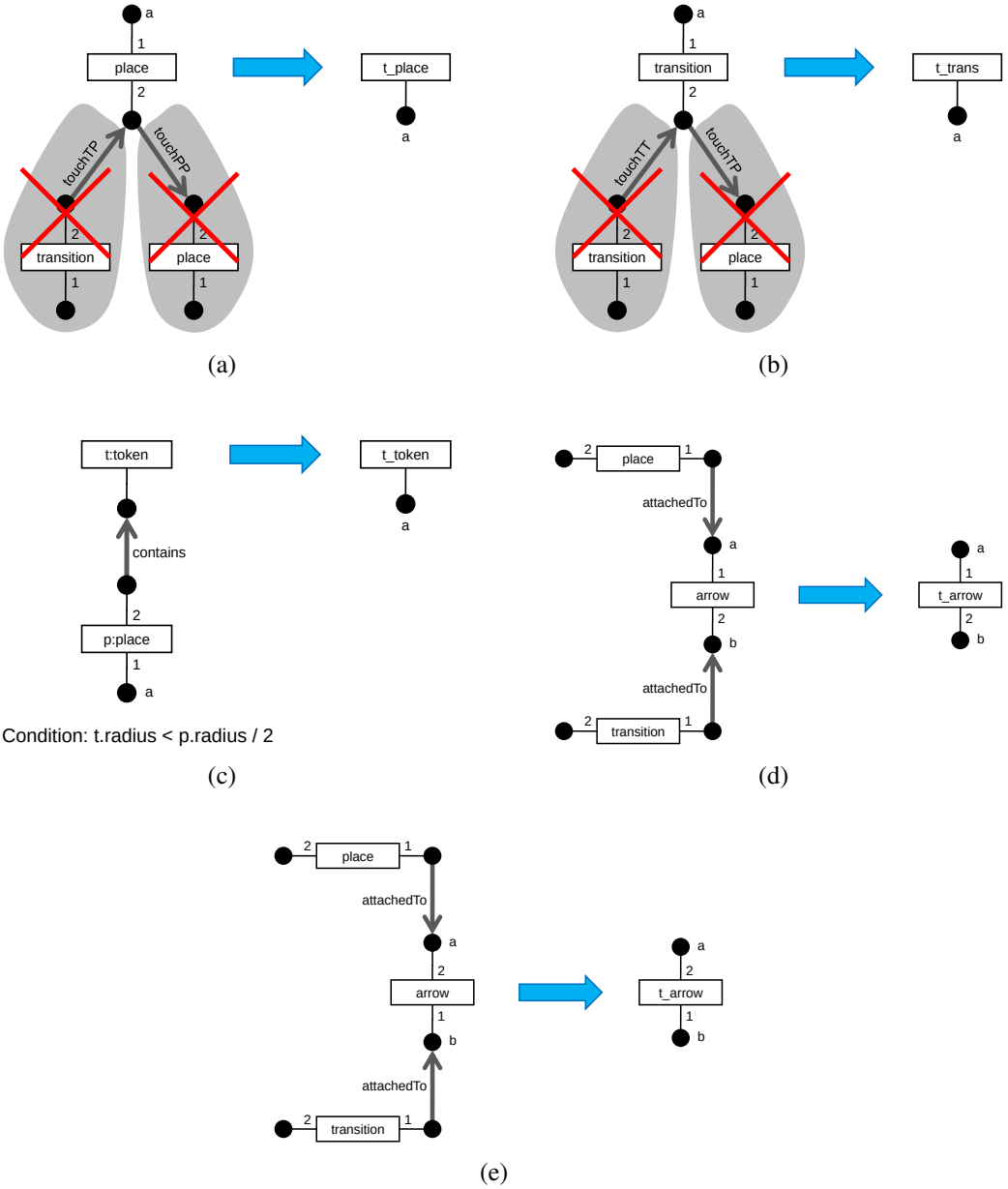


Figure 8.3: Reduction rules for Petri nets. Rules (d) and (e) are different in that the former corresponds to an arrow from a place to a transition, while the latter corresponds to an arrow from a transition to a place.

and token an attribute radius is defined, which contains the radius of the circle as value. Actions are given textually as well, but only refer to edges in the RHS (no actions are shown in Figure 8.3). Attributes of edges in the RHM have to be

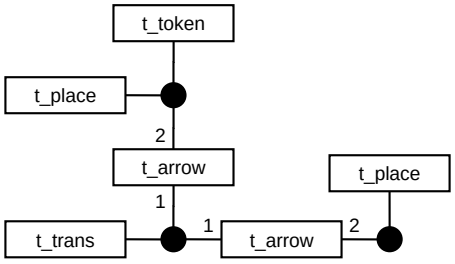


Figure 8.4: The RHM for the HM shown in Figure 7.7(b).

specified, too. Using the actions, their values can be set, and information can be forwarded from the HM to the RHM. Unlike those attributes of edges in the HM which can be generated and set automatically (cf. Section 7.4), for edges in the RHM all attributes have to be explicitly specified. NACs are shown as part of the LHS with a gray background, and crossed out (cf. 8.3(a), (b)). In this example, the NACs do not have conditions. Note that there are no reduction rules for arrows connecting two places or two transitions. These are invalid patterns which are not processed by the reducer, and are thus not present in the RHM.

As an example, using these rules the HM shown in Figure 7.7(b) is reduced to the RHM shown in Figure 8.4. In this case, the HM does not contain any touchXX edges, as no places and transition overlap in the sketch. It can be seen that the RHM is much smaller indeed, mainly because the relations are no longer explicitly represented by edges, and because there are far less nodes. In total, the HM consists of eleven edges and ten nodes, while the RHM consists of six edges and only three nodes. Reduction of the HM shown in Figure 7.7(c) will be discussed in the context of a larger example in Section 9.5.

Usually the design of reduction rules allows for some freedom. An alternative specification for Petri nets could use the rules shown in Figure 8.5 instead of rules 8.3(a) and 8.3(c), with all other rules from Figure 8.3 left unchanged. The difference is that tokens are no longer modeled explicitly by dedicated edges in the RHM, but by an attribute of edges labeled t_place. Accordingly, the rules in Figure 8.5 distinguish whether a place contains a token or not, and apply the correct action. In this example, there also is a condition for the NAC in Figure 8.5(a). This NAC prevents the application of the rule only if a match for the NAC is found *and* the token in the match is sufficiently small.

A special kind of reduction rule (which is not relevant to Petri nets) describes how nodes in the HM are merged in the RHM. This means that the reducer maps two nodes from the former model to only one node in the latter. An example is shown in Appendix A.2.8. This rule sometimes comes in handy (as for NSD, for example), and is another alternative to reduce the size of the RHM.

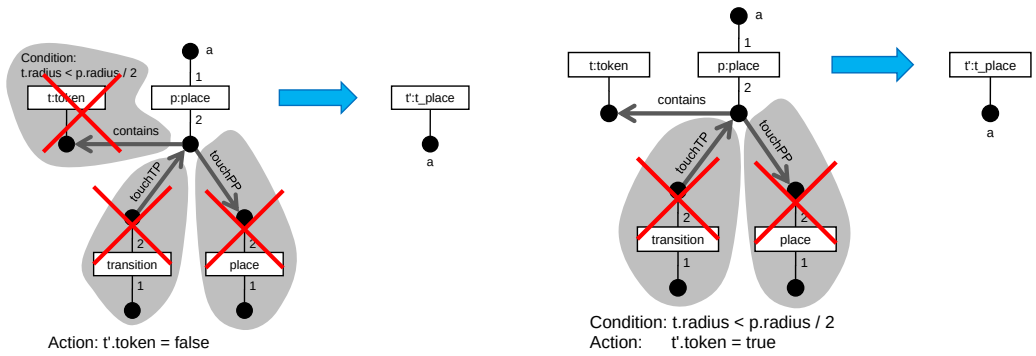


Figure 8.5: Two alternative reduction rules for Petri nets which are a substitute for the rules shown in 8.3(a) and (c).

8.3 Conflicts and Negative Application Conditions

The situation rarely is as simple as illustrated in the previous section. In the following paragraphs some further issues are therefore discussed. Handling these issues marks the difference between the reducer in DIAGEN and our adaption.

The first issue concerns the conflicts between shapes, which are identified in the postprocessing step (cf. Section 6.2). It may happen that shape edges matching an LHS of a reduction rule are conflicting. The rule must not be applied in this case, as a conflict indicates that one of the edges is wrong. The reducer therefore checks that no two conflicting edges occur in the same match for an LHS of a reduction rule. The same observation holds for NACs; a match for a NAC must not contain conflicting edges.

Taking a closer look at NACs reveals another issue. For example, the recognition stage identifies the transition and the place (or token) on the left hand side of Figure 8.6(a), next to the arrow and the transition on the right hand side. The modeler then creates the HM shown in 8.6(b). Now edges p and t mutually satisfy a NAC for each other, so for these two edges neither rule 8.3(a) nor rule 8.3(b) can be applied, although the LHSs of both rules are matched. The resulting RHM is shown in 8.6(c). At this point, all information about the place and transition is lost. Accordingly, the system can no longer recognize at least one of the two as correct. Obviously, it would be better if either the place or the transition could be included in the final result (although the sketch itself is syntactically wrong). Therefore, it is necessary that they are both represented in the RHM.

As a solution to this issue, the idea of conflicts has to be conveyed to the RHM. Like in the HM, edges in the RHM can also be *conflicting*, and again conflicts are a symmetric binary relation between these edges. Using conflicts, NACs can be

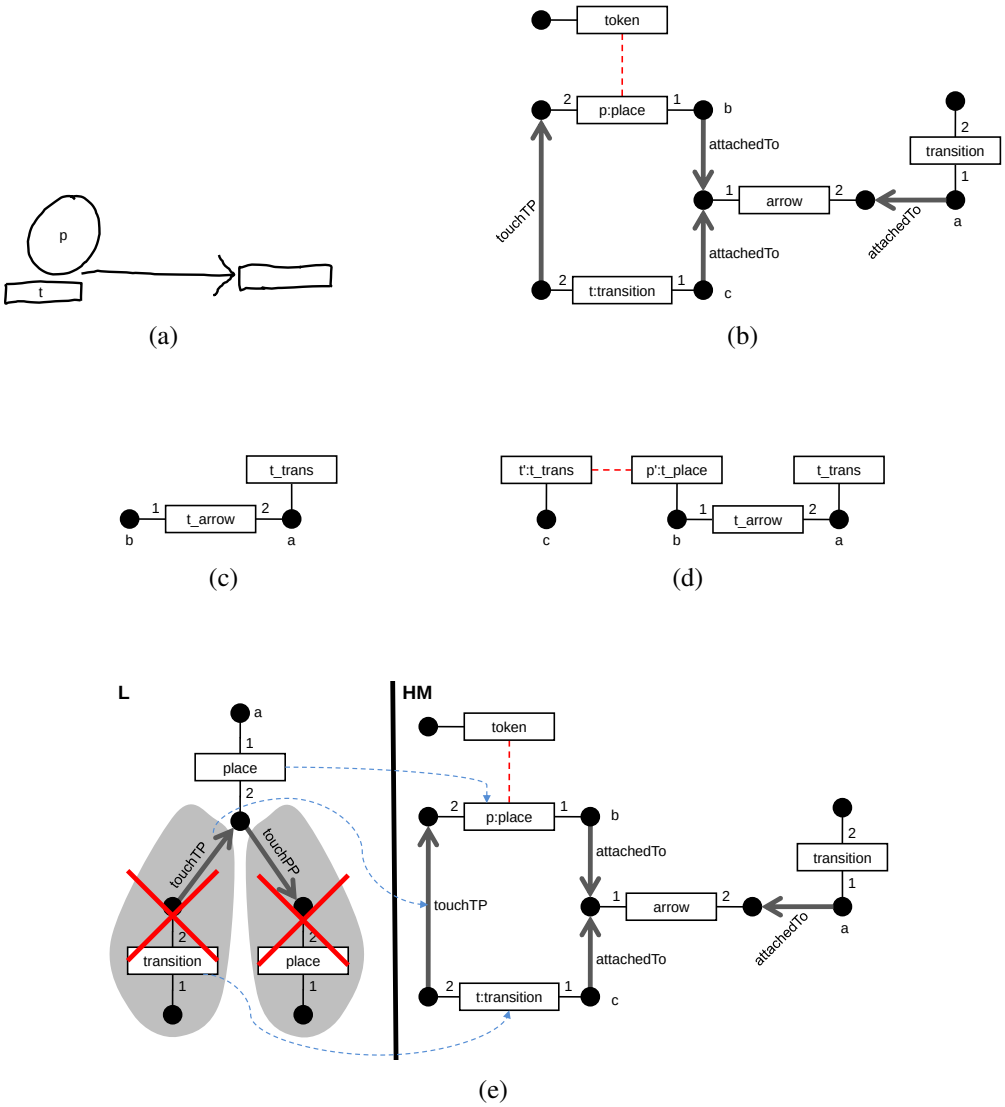


Figure 8.6: A sketched Petri net and different models created from the sketch. (a) hand-drawn Petri net. Place p and transition t touch each other, and both are attached to the arrow. (b) HM created for the Petri net. The token edge and the edge named p are not related because they are conflicting. (c) RHM if NACs are strictly obeyed. (d) RHM if processing of NACs is relaxed. (e) a match for the LHS and one NAC in the HM of the production rule shown in Figure 8.3(a).

processed differently. Instead of prohibiting the application of a reduction rule if a NAC is matched, the rule has to be applied anyway. In order to not render the NACs useless, respective information about the matched NAC has to be added to the RHM. This information can be expressed by conflicts, which is shown below. The desired result of this changed behavior of NACs is shown in 8.6(d) (note that there is, for (c) as well as for (d), no t_token edge generated, as in the HM in (b) the token edge conflicts with the place edge, which must not be the case for a match of the LHS, as discussed above).

Conflicts in the RHM, like the one shown in Figure 8.6(d), must be derived automatically from satisfied NACs. Informally speaking, if edges match a NAC in one application of a reduction rule, and match an LHS in the application of another rule, then there is a conflict in the RHM between the edges added from both rules' RHSs. In the example from Figure 8.6(b) there were two applications of reduction rules where a NAC was matched. With the changed behavior of NACs the following happens. The rule from Figure 8.3(a) is applied which reduces the place p to an edge labeled t_place, named p' in the figure. The touchTP relation and transition t satisfy a NAC for this application (cf. Figure 8.6(e)). The situation is very similar for the other rule that is applied, the one from Figure 8.3(b). It reduces the transition t to an edge labeled t_trans, named t' in the figure. The touchTP relation and place p satisfy a NAC for this application. Following the informal statement made in the beginning of this paragraph, both edges t' and p' must have a conflict in the RHM, and this is the case shown in Figure 8.6(d).

In both rule applications the touchTP relation edge was necessary to match a NAC. However, this edge is not in the match for an LHS of one of the rules. Still, a conflict emerges in the RHM. Accordingly, relation edges are not relevant when computing conflicts in the RHM, only shape edges are.

In the general case there can be more than just one shape edge in the match for an LHS and NAC. Then, it is more difficult to determine a conflict in the RHM. To express this circumstance precisely (and in preparation of the next chapter about the parser), two additional attributes are required for edges added to the RHM, the attributes shapeedges and nacs. The values of these attributes are set based on the rule applications. They work in the following way. Let there be a rule application r where

$$L = \{l_1, l_2, \ldots\} \text{ is the set of edges in the match for the LHS of } r$$

$$R = \{r_1, r_2, \ldots\} \text{ is the set of edges in the match for the RHS of } r$$

$$A = \{A_1, A_2, \ldots\} \text{ is the set of all matches of all NACs of } r$$

where each $A_i \in A$ is the set of all edges in one match of one NAC of r

In order to simplify the later definition of conflicts a function *shape* is required. Let S be a set of edges from the HM. Then

$$\textit{shape}(S) := \{s | s \in S \wedge s \text{ is a shape edge}\}$$

denotes that subset of S containing all shape edges in S. Then for all $r_i \in R$ the values of the two new attributes are set as follows.

$$r_i.\mathsf{shapeedges} = \textit{shape}(L)$$

$$r_i.\mathsf{nacs} = \{\textit{shape}(A_1), \textit{shape}(A_2), \ldots\}$$

For an edge e in the RHM both attributes describe which shape edges were required to create e, and which other combinations of shape edges were actually supposed to prevent this. Using these attributes, conflicts in the RHM can finally be defined. Let there be two edges e and e' in the RHM. Then

$$\mathsf{e} \text{ and } \mathsf{e'} \text{ are conflicting} \Leftrightarrow$$

$$(\exists N \in \mathsf{e.nacs} : N \subseteq \mathsf{e'.shapeedges}) \vee (\exists N \in \mathsf{e'.nacs} : N \subseteq \mathsf{e.shapeedges})$$

The last issue discussed in this section again refers to conflicts in the HM. In the following it is assumed that the different touchXX relation types introduced in Section 8.2 are left out from the specification, as well as all NACs requiring these relation types. Given that both a transition and a place (a token, respectively) are recognized in the sketch shown in Figure 8.7(a), the modeler creates the HM shown in Figure 8.7(b). Because there are no NACs for the reduction rules, the conflicting edges p and t are now valid matches for LHSs of reduction rules, resulting in a t_place edge and a t_trans edge in the RHM. In doing so, the original conflict from the HM would be lost (cf. Figure 8.7(c)). Consequently, the original conflict from the HM must be represented in the RHM, and again this can be accomplished with conflicts in the RHM (cf. Figure 8.7(d)). Informally speaking, if there is a match for the LHS in an application of a reduction rule, and edges conflicting with the edges from this match are themselves in a match for the LHS in the application of another rule, then there is a conflict in the RHM between the edges added from both rules' RHS. Using the concept established above, this can easily be expressed with the additional attribute nacs defined for edges in the RHM. As before, let there be a rule application r where

$L = \{l_1, l_2, \ldots\}$ is the set of edges in the match for the LHS of r

$R = \{r_1, r_2, \ldots\}$ is the set of edges in the match for the RHS of r

$A = \{A_1, A_2, \ldots\}$ is the set of all matches of all NACs of r

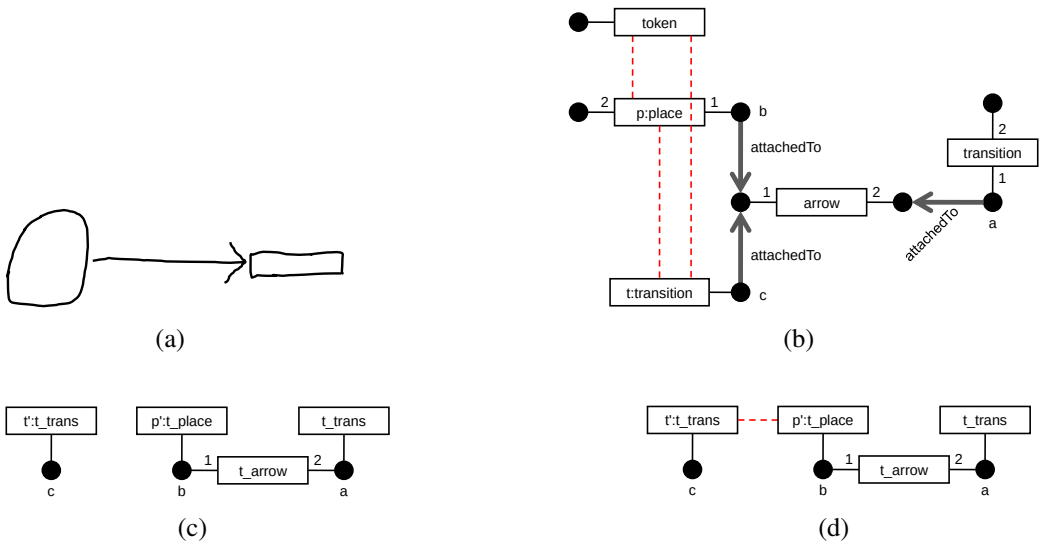


Figure 8.7: (a) a sketched Petri net. (b) HM if places and transitions may overlap. It is assumed that the recognizer identifies both a transition and a place (token) from the shape on the left hand side. (c) RHM where information about conflicts is lost. (d) RHM where information about conflicts is present. Note that the token is not reduced due to its conflict with the place edge in the HM.

where each $A_i \in A$ is the set of all edges in one match of one NAC of r

Let C be the set of all shape edges in the HM which have a conflict with an edge in L, i.e.,

$$C = \{c_1, c_2, \ldots\} :=$$

$$\{e | e \text{ is a shape edge in the HM} \wedge \exists l \in L : e \text{ and } l \text{ are conflicting}\}$$

Using this definition, C cannot contain relation edges, as conflicts occur only between shape edges in the HM. Now for all $r_i \in R$ the value of attribute shapeedges is set as before, and the value for attribute nacs is set as follows.

$$r_i.\text{nacs} = \{\mathit{shape}(A_1), \mathit{shape}(A_2), \ldots\} \cup \{\{c_1\}, \{c_2\}, \ldots\}$$

This example shows that conflicts in the HM are automatically propagated to the RHM, even if there are no suitable NACs. It can also be seen that conflicts in the HM, as well as matches for NACs, are mapped to the same concept in the RHM. In the RHM it does not matter from what circumstance a conflict emerged. The parser, discussed in the next chapter, processes these conflicts.

In Section 5.5 ratings have been introduced for shapes. These ratings have to be carried over to the RHM in order to influence the parsing process. We define

the rating of an edge t in the RHM as

$$\mathsf{rating}(\mathsf{t}) := \sum\{\mathsf{rating}(\mathsf{e}) | \mathsf{e} \in \mathsf{t.shapeedges}\}$$

NACs, conditions and relation edges do not influence the rating of t.

8.4 Future Work

Interpreting NACs in a way that they do not prevent application of reduction rules, but have extra information generated into the RHM as explained in Section 8.3, has proven to be valuable for DSKETCH. The conflicts which are added to the RHM are considered by the parser to compute only syntactically valid derivation structures. However, regular diagram editors not based on sketching, but on traditional user interfaces, may also benefit from this changed interpretation. The DIAGEN system is an example. The actual gains possible by interpreting NACs differently have to be investigated. A possible benefit may be that the system could show more precise error messages in case of syntactically erroneous diagrams.

8.5 Summary

Based on the HM created by the modeler, the reducer creates the RHM. It is usually smaller than its predecessor in terms of number of edges and nodes. This allows for a more efficient parsing. Invalid patterns in the HM are ignored by the reducer, and do not have to be taken into account by the parser. The structure of the HM is no longer domain-independent as the HM was, but depends on the domain.

The reduced hypergraph contains information about conflicts between edges. This information is generated either from conflicts between edges of different matches in the HM, or by the application of NACs. The key idea of NACs is to apply rules even if matches for NACs would actually prevent this. Then, the parser can use the identified conflicts when processing the RHM.

Chapter 9

Parser

The final step in the processing of a hand-drawn diagram is the parsing of the RHM (cf. Figure 1.4). This allows both for checking the syntax of the diagram and for generating a derivation structure which can then be used to determine the semantics of the diagram. The applied parser is a bottom-up parser which treats the edges in the RHM as terminal symbols. For this reason the names of all edges in the RHM start, by convention, with t_ to indicate their role. According to the grammar and its production rules given in the specification, the parser tries to deduce the start symbol of the grammar. If this succeeds, the semantics of the drawing can be evaluated based on the derivation structure. If deduction of the start symbol is not possible, no semantics can be computed, and the drawing is found to be incorrect. The parser works on a best-effort basis. If it is not possible to deduce the start symbol using all terminal symbols, a subset is considered, while the remaining symbols are ignored. The subset is chosen based on the ratings of the terminal symbols. The derivation structure is, depending on the productions, either a tree, or a directed acyclic graph (DAG).

The parser works similar to a Cocke-Younger-Kasami (CYK) parser [1] for string grammars. Consequently, the hypergraph grammar must be transformed into Chomsky normal form (CNF). The consequences of this approach are investigated in Section 9.1. As stated in the previous chapter, the parser must regard the conflicts found between edges in the RHM. This is necessary for every deduction step, and defines the difference to the original DIAGEN approach. The basic idea is to forbid the deduction of nonterminal symbols if this involves using conflicting terminal symbols. The detailed discussion of this idea is split according to the different kinds of production rules, and can be found in Sections 9.2 through 9.4. A larger example illustrating the interplay of the rules is given in Section 9.5. Semantics based on attribute evaluation are explained in Section 9.6, along with an illustration of which diagram is chosen if there is a selection of several possible diagrams. Finally, Section 9.7 summarizes the chapter.

Production rules have an LHS and an RHS. In the following we say that production rules are *applied*. The application of a production rule is a *production*. The LHS of a production (rule) is said to be *derived* into the RHS, or the LHS is said to be *deduced* from the RHS, which means the same, but reflects better that the parser works bottom-up.

9.1 Production Rules

A CYK parser allows for parsing strings according to a context-free string grammar. As a premise, the grammar must be transformed into CNF, which is possible for context-free grammars if and only if the empty string is not in the language of the grammar. DIAGEN adopts this approach, and allows for parsing context-free diagram languages similar to CYK. The difference lies in the relation between terminal symbols. In string grammars, terminal symbols are given as a sequence, each having one predecessor (except for the first one), and one successor (except for the last one). In diagram languages, terminal symbols (represented as hyperedges), do not form a sequence. Each terminal symbol is related to a fixed number of other terminal symbols by visiting the same nodes (the actual number depends on the label of the terminal symbol). It is not meaningful to speak of a predecessor or a successor in this case.

As has been shown in [68], context-free hypergraph grammars can also be transformed into CNF, and the CYK approach can be applied similarly. This means that there are only two different kinds of production rules to be considered:

- A nonterminal symbol that is derived into two nonterminal symbols (a nonterminal production rule, abbreviated NPR).
- A nonterminal symbol that is derived into a terminal symbol (a terminal production rule, abbreviated TPR).

We omit rules where the start symbol is derived into the empty diagram. The production rules for Petri nets are shown in Figure 9.1, before the transformation to CNF. Hereby it is assumed that tokens are represented explicitly by edges in the RHM (as in Figure 8.3). Production rules for arrows are not discussed now, but below.

Edges from the RHM are called *terminals* in the following. Nonterminal symbols, which are hyperedges as well, are called *nonterminals*. The labels of nonterminals begin with an uppercase letter, again by convention. Furthermore, Figure 9.1 shows nonterminal symbols as rounded rectangles, and so all following figures will do.

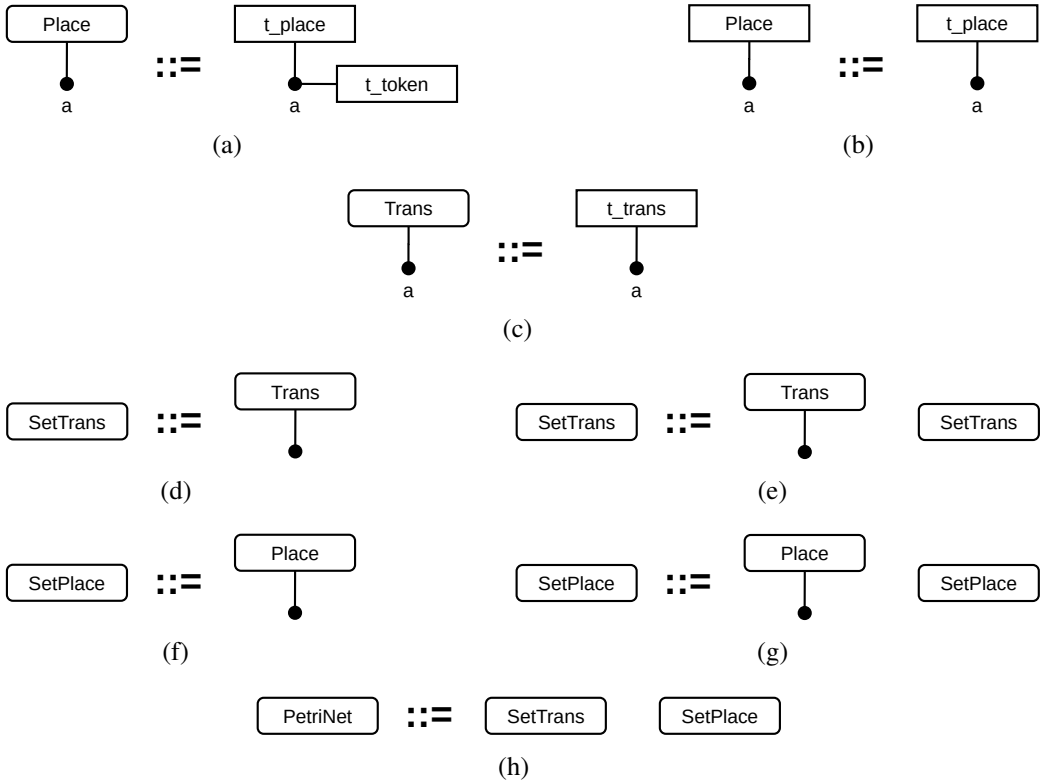


Figure 9.1: Production rules for Petri nets. Rules for arrows are not shown. The start symbol is labeled PetriNet.

As can be seen from the rules, the grammar for Petri nets exhibits five different nonterminals labeled PetriNet, SetPlace, SetTrans, Place, and Trans. In this simple example, only edges labeled Place or Trans visit a node, all other nonterminals do not. Note that it would actually be sufficient for Place and Trans edges not to visit any nodes as well, as these nodes are not referred to afterward. However, they both do, in order to also show nonterminal edges visiting nodes in this example.

The meaning of the rules is straightforward. The start symbol PetriNet is derived into a set of places SetPlace and a set of transitions SetTrans (h). SetPlace is either a single Place (f), or another SetPlace and a Place (g). The same holds for a set of transitions SetTrans, (d) and (e). A Place is a terminal t_place, either with (a) or without a t_token (b). A Trans simply is a terminal t_trans (c).

The two rules creating a set of places (f) and (g) closely follow a pattern well-known from string grammars (the same is true for rules (d) and (e) creating a set of transitions). A sequence A of terminal or nonterminal symbols a in a string grammar is usually obtained by two rules like A ::= a | a A which effectively

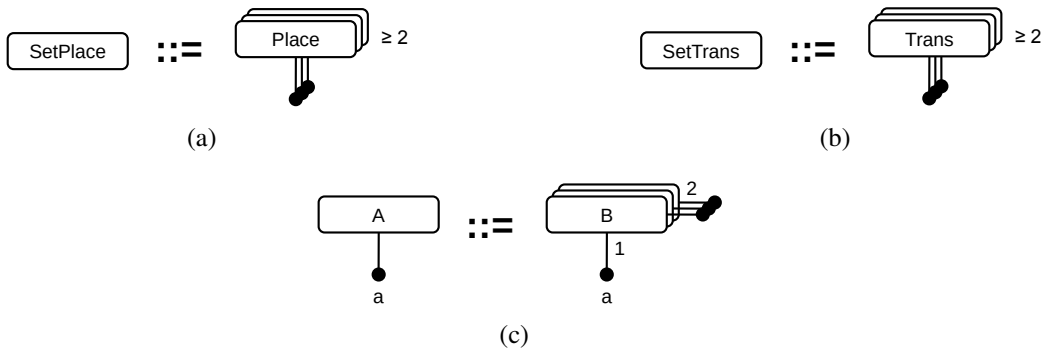


Figure 9.2: (a), (b) set production rules for Petri nets. (c) generic rule not part of the grammar for Petri nets (each edge in a match of the RHS must visit the same node a, and an arbitrary second node).

derive the symbols. For graph grammars the case is different. On the one hand, the two pairs of rules mentioned above indeed collect all places and transitions. On the other hand, as there is no ordering on the symbols, the set of all places is deduced many times, one time for each possible order of all places. This does not happen with string grammars and severely hurts performance, as we are not interested in any order, but just in a *set* of all places. However, the issue is even more severe, as not only each possible ordering of all places is deduced, but also each possible ordering of each possible subset of all places. This results in a combinatorial explosion which must be avoided.

To overcome this issue a new kind of production rule has been introduced in DIAGEN, which is called *set production rule* (SPR) [70]. A set production rule derives from a nonterminal the largest possible *set* of other nonterminals, each with the same label, not regarding any order. The minimum number of nonterminals in the set is 1 by default; however, a larger number may be specified. There are no other edges in the RHS of an SPR. Figures 9.2(a) and (b) show two set production rules which replace the four rules shown in 9.1(d) through (g). Graphically, set production rules are always shown by a stack of edges in the RHS of the rule. In this example, the minimum number of nonterminals for both productions is set to 2, meaning that there must be at least two places and two transitions. When deriving set productions, the parser still checks nodes visited by the edges. All edges in the RHS can be required to visit the same node, or each edge may visit an arbitrary node. The latter is graphically indicated by a stack of nodes and tentacles as shown in 9.2(a) and (b). The former is indicated by just one node and tentacle. A generic example for a set production is given in Figure 9.2(c). Nonterminal A visits node a that is also visited by *all* edges B in the set. Additionally, all these edges visit a second node, which does not need to be equal for all edges.

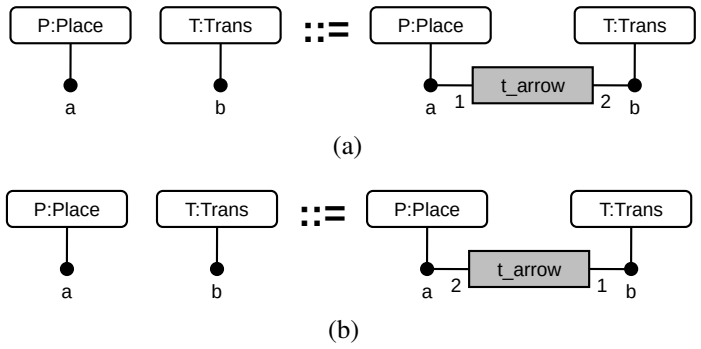


Figure 9.3: Embedding production rules for Petri nets.

The problem with the arrows, which were omitted before, is that arrows cannot always be described by context-free production rules. For diagrams with a tree-like structure it is possible, for diagrams with a graph-like structure (like Petri nets) it is not. A simple binary tree consisting of nodes connected by arrows can indeed be specified using context-free rules only, for example. The presence of non-context-free rules usually requires a parser with a higher complexity than for the context-free case. However, DIAGEN has introduced a special kind of production rule which allows for embedding arrows (and other shapes alike) in a given context with acceptable costs. This kind of rule is the *embedding production rule* (EPR), consisting of the same non-empty hypergraph in the LHS and the RHS (called the *context*), with one additional edge in the RHS which is to be embedded into the context (the *embedded edge*). It is important that the derivation of the context must use context-free rules only. As arrows connect either a place to a transition, or a transition to a place, two embedding production rules are required. Both are shown in Figure 9.3. The embedded edge on the right-hand sides of both rules is shaded in gray. It is important to give names to all nodes and edges of the context both in the LHS and the RHS to account for a unique correspondence between LHS and RHS. Note that it would also be possible to embed t_arrow edges into a context of a t_place edge and a t_trans edge.

While the parser deduces nonterminals in order to finally deduce start symbols, contexts for embedded edges are deduced as well. After this, for each deduced start symbol S, each embedded edge e is added to the derivation structure with root S if all edges from the context of e are in the derivation structure as well.

All kinds of production rules (NPR, TPR, SPR, and EPR) can have an optional condition. If this condition is specified, it is checked if it holds before the rule is applied. Conceptually equal to the conditions of reduction rules, conditions of production rules may refer to nodes, edges, and attributes of the edges.

9.2 Terminal and Nonterminal Production Rules

In this section and the following two it is discussed how conflicts are treated by the parser. First some definitions are required. Two non-empty sets A and B of terminal symbols are said to be *conflicting* or *to have a conflict* if there is a terminal in A that conflicts with a terminal in B. Furthermore, each terminal or nonterminal e is assigned a set of terminals, referred to as term(e). For a terminal e we define term(e):={e}, for a nonterminal e we define that term(e) contains exactly those terminals which are derived from e in one or more productions. Finally, two terminals or nonterminals e and e' are *conflicting* if term(e) and term(e') are conflicting.

The general principle which must hold for all kinds of production rules for the deduction of nonterminals is the following: no nonterminal must be deduced from two conflicting symbols in the match for RHS of a production rule. The direct consequence of this requirement is that every nonterminal is always derived into a set of terminals, where no two terminals in the set are conflicting. Hence, this also holds for the start symbol, which means that indeed only legal diagrams are derived from a start symbol, i.e., diagrams without conflicts. The only conflicts that may occur (and do in practice) are those between terminals which are not derived from the same nonterminal.

For terminal and nonterminal production rules (TPRs and NPRs) the situation is simple. A TPR may always be applied if its condition holds. As there is only one edge in the RHS of the rule, no conflicts can arise.

For NPRs with two nonterminals r and r' in the RHS, the condition of the rule must hold, and r and r' must not be conflicting. This can be checked easily, given that term(r) and term(r') are known, which is just a matter of bookkeeping: Let l be the nonterminal in the LHS of the production rule deriving r and r'. Then term(l) is defined as the union of term(r) and term(r'). For TPRs with terminal e in the RHS, where l is again the nonterminal in the LHS, term(l) is given by term(e).

For NPRs, there is another issue which must be checked. For the two nonterminals r and r' in the RHS of an NPR, term(r) and term(r') must be disjoint, because no terminal must be derived from two different nonterminals. This condition is known from string grammars as well. Therefore, the notion of a *conflict* between nonterminals is extended, and two nonterminals r and r' are called *conflicting* if either term(r) and term(r') are not disjoint, or if term(r) and term(r') are conflicting.

9.3 Set Production Rules

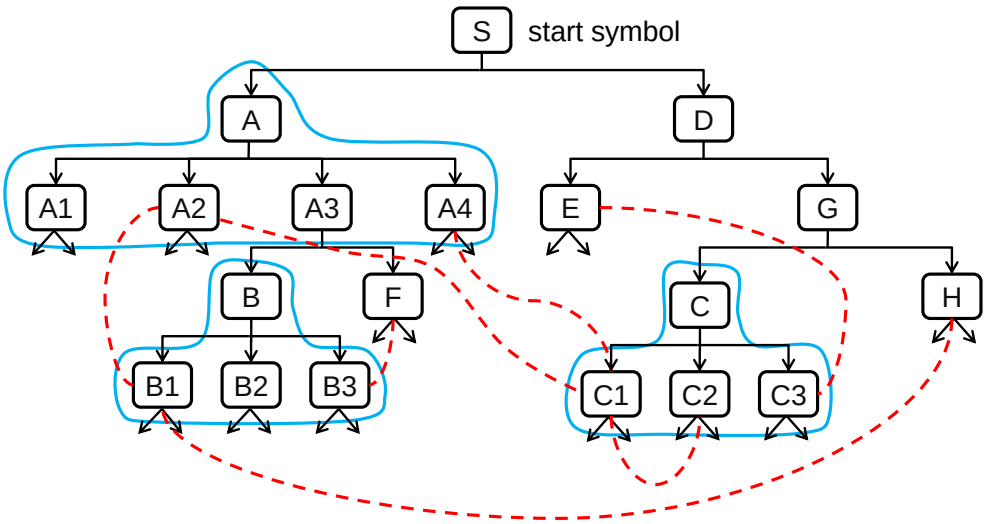
As described before, a set production rule (SPR) derives a nonterminal in the LHS to an unordered set of (non)terminals in the RHS [70]. Each edge in the RHS has the same label and, if specified, visits certain nodes. A minimum number of edges required in the set can be specified, which prevents derivation of the LHS if this number is not satisfied. Like all other production rules, a set production rule may also have a condition.

Deduction of an SPR is similar to that of a nonterminal production rule. The parser identifies all edges matching the RHS of the SPR. If the number of these edges exceeds the required minimum, and if the condition of the rule holds, the rule may be applied. As shown for nonterminal production rules in the previous section, no two nonterminals may be deduced if they are conflicting. This crucial condition must be satisfied as well when applying an SPR. The consequence is that of each two conflicting edges in the set in the RHS, one must be left out. However, in general there are further deductions necessary before the start symbol is reached. Leaving out the wrong one of the two edges can lead to unforeseeable consequences in the later steps.

As an example take a Petri net with several places, two of which are conflicting. One of them, p1, has an arrow attached to a transition, the other, p2, has not. p1 seems to be more valuable in this case, and should not be left out. If it would, the arrow could not be embedded. However, as embedding productions are applied last, the parser does not know about any embedding productions when deducing the set production rule. The same is true for conflicts which may arise in later applications of nonterminal production rules.

The solution is to postpone the decision about leaving out edges from the set until the start symbol has been deduced. This solves the aforementioned issue (even for embedding production rules, which are discussed in the next section). After the start symbol has been deduced, all context information in terms of syntactic structure is obtained, and reasonable decisions about leaving out symbols can be made. Under certain circumstances, however, edges can actually be left out before this. Figure 9.4 shows an exemplary derivation tree, and lists the applied productions. The text written inside each edge is not a label, but a unique name; the labels of the edges are of no interest in this example. Nodes are not shown. Set productions are encircled, two arrows leaving an edge and ending in no other edge mean that the derivation tree continues here, which is also of no interest for the example. As before, conflicts are shown by dashed lines.

After A3 has been deduced from B and F, it can be seen that B3 must be immediately removed from its set, as it conflicts with F. F cannot be left out, as it is in no set. This is the simplest case when an edge in a set can be left out even before the start symbol is deduced. Accordingly, after D has been deduced from



Nonterminal productions:

S → A D
A3 → B F
D → E G
G → C H

Set productions:

A → A1 A2 A3 A4
B → B1 B2 B3
C → C1 C2 C3

Figure 9.4: Exemplary derivation tree. Conflicts are shown as dashed lines.

E and G, C3 must be immediately removed, although it is not G which has been deduced in a set production, but C. The third case regards the start symbol S being deduced from A and D. Due to the conflict between H and B1, it is B1 which must be left out.

In general, after an NPR is applied, conflicts between edges in sets and edges not in sets can be immediately solved by removing those edges which are in sets. This must also happen if set productions are cascaded, as seen in the example. It may happen that a set has left too few edges according to its defined minimum after the NPR is applied. If the set is not in another set (like the set derived from A, and the set derived from C) the NPR must not be applied. If the set is in another set (like the set derived from B), the respective edge must be removed from its containing set. In the example, A3 must be removed from its set if the set derived from B contains fewer edges than its minimum size.

Assuming that each of the three set productions is satisfied with one edge, the derivation tree shown in Figure 9.5 is obtained after the start symbol is deduced. Due to the removal of edges in sets in case of conflicts with edges not in sets

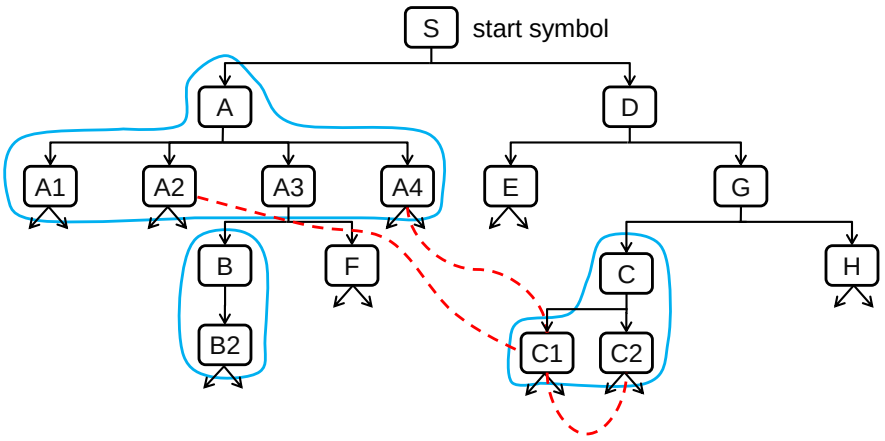


Figure 9.5: Exemplary derivation tree after some conflicts are solved.

(as described above), all remaining conflicts are between edges in sets, in this example A2, A4, C1, and C2. In this case it seems to be most valuable to remove C1, as this solves all conflicts at once. However, we propose a more sophisticated solution, which is described in the following. Note that all edges in sets which do not have any conflicts (in this example A1, A3, and B2) can be ignored, as there is nothing to do here.

We define the rating rating(n) of an nonterminal n as the sum of all ratings of the terminals in term(n), i.e.,

$$\mathsf{rating}(\mathsf{n}) := \sum \{\mathsf{rating}(\mathsf{t}) | \mathsf{t} \in \mathsf{term}(\mathsf{n})\}$$

For the start symbol, the rating is to be maximized, as ratings are defined in a way that a greater rating corresponds to a better solution. Accordingly, the task is to find a subset of the conflicting edges in the sets such that no two edges in the subset are conflicting, and such that the accumulated rating of all edges in the subset is maximized. We solve this optimization problem by a heuristic, as it is NP-complete. Taking a quick glance at graph theory reveals the *clique problem*, which was found to be NP-complete by Karp in 1972 [60]. Given an undirected graph G (not a hypergraph), the clique problem is determining whether G contains a clique of at least size k. A clique in G is a complete subgraph of G, i.e., each node in the clique is adjacent to each other node in this clique [41]. The optimization problem linked to the clique problem is to determine the largest clique in G.

Applied to our problem of conflicting edges in sets we can define G as a graph where each conflicting edge from a set is a node, and where two nodes are adjacent if and only if their corresponding edges are not conflicting. The graph G for the example in Figure 9.5 is shown in Figure 9.6. If we set all ratings for all edges

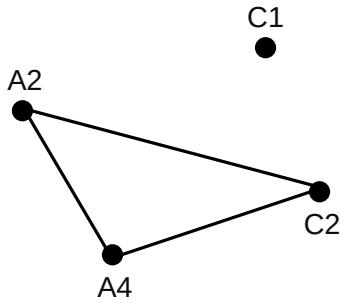


Figure 9.6: Graph G, constructed from the derivation structure in Figure 9.5. Each node in G corresponds to an edge in Figure 9.5 with a conflict. Nodes are adjacent if the two corresponding edges are not conflicting. The largest clique is searched for.

to 1, the solution to our problem also solves the clique problem. As the latter is known to be NP-complete, our problem is as well.

Now the idea of the heuristic we apply to this problem is to prefer edges (i) with a high rating, (ii) with few conflicts, and (iii) where the conflicting edges have low ratings (in the next section there will even be a fourth aspect related to embedding production rules, which we omit here). For each edge e we define the weighted rating $\omega(\mathsf{e})$ as

$$\omega(\mathsf{e}) := \begin{cases} \frac{\mathsf{rating(e)}}{\sum\{\mathsf{rating(c)} | \text{edge c has a conflict with e}\}} & \text{if e has conflicts} \\ \infty & \text{otherwise} \end{cases}$$

Obviously, the weighted rating of an edge is higher the better (i)–(iii) are satisfied. Then the edge l with the lowest weighted rating is removed, and the weighted ratings of all edges which had a conflict with l are updated, as they have one less conflict now. Then again the edge with the lowest weighted rating is removed, and all affected weighted ratings are updated, as long as there still is a conflict.

What can happen is that an edge is removed such that its set has too few edges left and does not satisfy its minimum any longer. Then again a distinction must be made whether the set is in another set, or not. In the former case, the respective edge in the parent set is removed as well, which again may lead to the case that the parent set does not satisfy its minimum any more. In this case, the removal has to be repeated. In the latter case where the set is not in another set, the removal must be undone, and the edge with the second lowest rating is tried. This in effect is backtracking. The first solution which is found by this heuristic is immediately accepted. If there cannot be found any solution, the start symbol is invalid, and must not be regarded for further processing.

As an example, consider again Figure 9.6. Given that the rating of C1 is 3.0, and the ratings of A2, A4, and C2 are 2.0, the weighted rating of C1 is 3/6, and

the weighted ratings of the other edges are 2/3, so C1 is discarded first, and the optimal result is obtained.

Given that the rating of C1 is 2.9, and the ratings of A2, A4, and C2 are 1.0, the weighted rating of C1 is 29/30, and the weighted ratings of the other edges are 10/29, so one of them is removed first. The rating of C1 changes to 29/20, and again one of the other edges is removed. Finally, the third of those is removed, too, and only C1 remains. This is not the optimal solution, but still a very good one, according to the ratings (2.9 compared to 3.0).

To conclude this section, conflicts within sets obtained by set productions are ignored first. Conflicts between edges in sets and edges not in sets can be solved immediately. After the start symbol is deduced, the remaining conflicts are also solved based on a heuristic.

9.4 Embedding Production Rules

Embedding production rules (EPRs) are not considered before the start symbol has been deduced. Then, for all EPRs the contexts are searched for in the derivation tree. For each match of a context, the respective edge is embedded in the tree (as indicated by the rule), if there is a respective edge present. For EPRs where the embedded edge is not a terminal, but a nonterminal, this nonterminal is regarded as start symbol for the EPR and deduced like the regular start symbol of the grammar. This way, the embedded edge is always valid, as there is no conflict in its terminal edges. What remains to be checked if an edge ee which is to be embedded is that ee is not conflicting with the start symbol of the derivation tree (which includes that there is no conflict with the context of the embedded edge, and that ee is not conflicting with any other edge embedded before). If the latter happens, that edge which has the higher rating is preserved, and the other edge is discarded. By embedding edges in the derivation tree, the tree becomes a directed acyclic graph (DAG). The rating of the start symbol is increased for every embedded edge by the rating of that edge.

Obviously this approach can also be applied when set productions are present. However, when conflicts in sets are resolved by the heuristic explained in the previous section, additional care has to be taken for the embedding of edges. A conflicting edge c is more valuable if it is part of a context for an embedded edge ee. If c is removed, ee can no longer be embedded and the rating of the start symbol decreases. Accordingly, the computation of the weighted rating of an edge must be modified in order to also prefer those edges (iv) which are part of a context for an embedded edge, or which have a child that is part of such a context. (i)–(iii) remain as described before. Let e be an edge, and context(e) be the set of all embedded edges where e or a child of e is part of their context. Then the

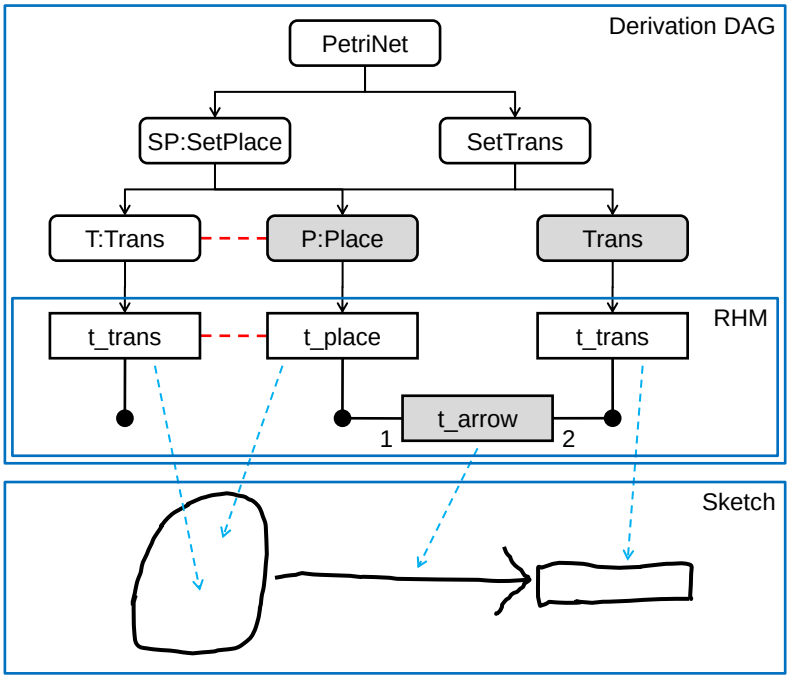


Figure 9.7: A sketch and the generated RHM and derivation DAG. The dashed arrows indicate which terminals represent which shapes. The embedded t_arrow edge and its context are shaded.

weighted rating $\omega(\mathsf{e})$ of e is computed as

$$\omega(\mathsf{e}) := \begin{cases} \frac{\mathsf{rating(e)} + \sum\{\mathsf{rating(ee)} | \mathsf{ee} \in \mathsf{context(e)}\}}{\sum\{\mathsf{rating(c)} | \text{edge } \mathsf{c} \text{ has a conflict with } \mathsf{e}\}} & \text{if } \mathsf{e} \text{ has conflicts} \\ \infty & \text{otherwise} \end{cases}$$

The heuristic works as before, but whenever a conflicting edge c is removed (because it has the lowest weighted rating), not only the weighted ratings of those edges must be updated which had a conflict with c, but also those which were part of the same contexts, as the respective embedded edges can no longer be embedded.

As an example we pick up the sketch shown in Figure 8.7(a) and its RHM shown in Figure 8.7(d). These are also shown in Figure 9.7, together with the derivation DAG generated by the parser. After the start symbol is deduced, one conflict remains between edges T and P. As example calculation it is assumed that all terminals in the RHM have the same rating, say 2.0. Then, the weighted rating of T is $2.0/2.0 = 1$, and the weighted rating of P is $(2.0 + 2.0)/2.0 = 2$. $\omega(\mathsf{P})$ is greater than $\omega(\mathsf{T})$ as P is in the context of the arrow. Consequently, T is discarded, and the desired result is obtained.

In another example calculation different ratings are assumed for the terminals, which leads to a configuration that requires backtracking. Given that the t_trans edges in the RHM each have a rating of 3.0, and the two other edges in the RHM have a rating of 1.0, the weighted rating of T becomes $3.0/1.0 = 3$, while the weighted rating of P becomes $(1.0 + 1.0)/3.0 = 2/3$. As this rating is lower, P will be removed to resolve the conflict. However, this results in SP having no more children, so the removal must be undone, and the edge with the second lowest weighted rating is removed, which is T.

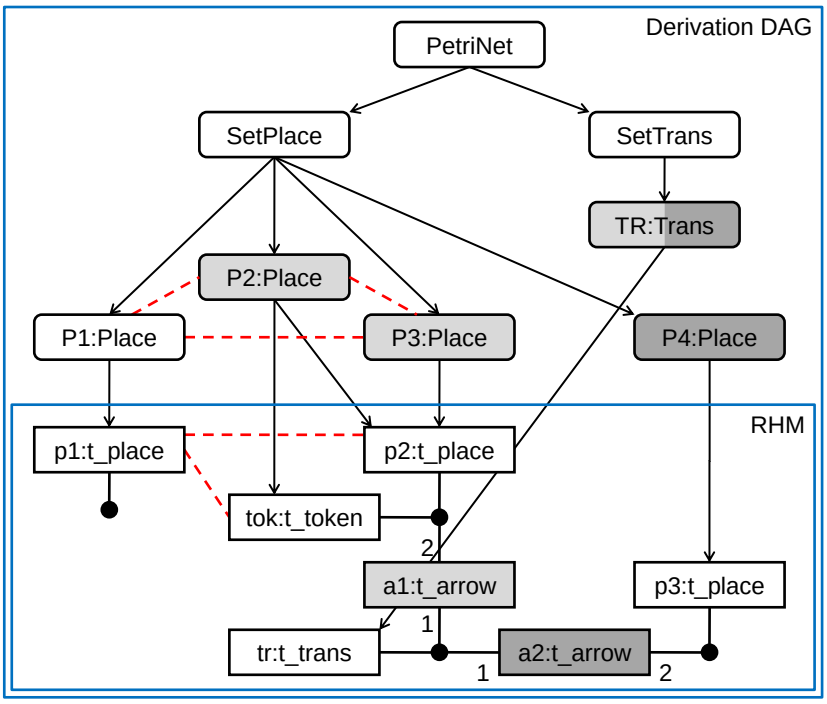
9.5 A Larger Example

In Figure 7.7(a) a sketched Petri net has been shown with two places, one token, one transition, and two arrows. The HM generated for this sketch is shown in Figure 7.7(c). We assume the reduction rules in Figure 8.3. Using these reduction rules, the RHM shown in Figure 9.8(a) is obtained. There are two conflicts between terminals. The conflict between p1 and tok emerges from the conflict in the HM, and the conflict between p1 and p2 emerges from the fact that the corresponding places are touching (not shown in Figure 7.7(c)). Note that tokens not contained in places are not reduced. Furthermore, for the only non-omitted contains relation edge in Figure 7.7(c), no t_token edge is reduced, as the token has a larger radius than the place, which contradicts the condition of the reduction rule in Figure 8.3(c).

Figure 9.8(a) also shows the derivation DAG generated by the parser, assuming the production rules established throughout this chapter (the ones shown in Figure 9.1(a), (b), (c), (h), the set production rules shown in Figure 9.2(a), (b), and the embedding production rules shown in Figure 9.3(a), (b)). Except for P4, all Place edges are conflicting. P2 and P3 are conflicting as they are derived into the same terminal p2. The two other conflicts emerge from the conflicts between the terminals. For a1 there are two contexts possible, TR and P2, or TR and P3. For a2 there is only one context, TR and P4. Note that the two contexts for a1 could also be avoided by using the terminal edges as contexts, as opposed to the nonterminals as in (cf. Figure 9.3).

Using the weighted ratings and the heuristic, the conflicts between P1, P2, and P3 are resolved. Obviously only one of the three nonterminals survives this process, as each has a conflict with each other. The expected survivor is P2, as this nonterminal represents the outer circle being a place, and the inner circle being a token, and allows for embedding of a1.

Example ratings of the terminals are shown in Figure 9.8(b), as well as the ratings for the nonterminals. Based on these ratings, the weighted ratings for the three conflicting nonterminals can be computed, all of which are in the same set.



(a)

	Terminals							Nonterminals				
Name	p1	p2	p3	tr	tok	a1	a2	P1	P2	P3	P4	TR
Rating	3.9	3.8	4.1	4.0	3.9	2.7	2.6	3.9	7.7	3.8	4.1	4.0

(b)

	P1	P2	P3
Initial weighted ratings	$\frac{3.9}{7.7+3.8} \approx .34$	$\frac{7.7+2.7}{3.9+3.8} \approx 1.35$	$\frac{3.8+2.7}{7.7+3.9} \approx .56$
P1 is removed		$\frac{7.7+2.7}{3.8} \approx 2.74$	$\frac{3.8+2.7}{7.7} \approx .84$
P3 is removed		∞	

(c)

Figure 9.8: (a) RHM for the HM from Figure 7.7(c), and its derivation DAG. Embedded edges and their contexts are shaded. The reduction rules from Figure 8.3 are assumed. (b) ratings of terminals and nonterminals. (c) initial weighted ratings and updates based on removal of conflicting edges.

Figure 9.8(c) shows these initial weighted ratings, and updated weighted ratings after nonterminals are removed. Each row in the table represents the removal of one nonterminal, until indeed P2 remains.

9.6 Attribute Evaluation

With one minor exception (see below), attribute evaluation works in DSKETCH just like it does in DIAGEN. When specifying the grammar, each nonterminal may be given attributes, depending on the label of the nonterminal, just like the attributes of the shape edges used by the modeler (cf. Chapter 7), and attributes of the terminals set by the reducer. The start symbol is required to have one distinguished attribute, the *semantic attribute*. Given a derivation DAG (or tree), its root always is a start symbol. The final value of the semantic attribute of the root is considered as the semantics of the DAG. It follows that the grammar is an attributed grammar. Just like for reduction rules, each production rule, no matter what kind, has an action which allows for setting the attributes of the nonterminal(s) in the LHS based on the attributes of the edges in the RHS. Examples for actions of production rules can be found in Appendix A.

If the value of the semantic attribute of the root cannot be evaluated for some reason, then the shapes represented by the DAG are syntactically correct, but considered semantically wrong. Hence this DAG is no valid interpretation for the sketch (or part of the sketch).

Depending on the grammar there may be different derivation DAGs deduced by the parser. For Petri nets this cannot happen, but for NSD, for instance, it can. Unlike DIAGEN, in this case that DAG is chosen to represent the sketch where the root has the highest rating and the semantic attribute can be evaluated. All other DAGs are discarded. The rationale behind this procedure is the following: two different DAGs always represent two different interpretations of the sketch, and occur because different sets of derivations led to the start symbol. For example, the parser tries to derive the start symbol with each of two conflicting edges, albeit not in the same derivation structure, of course. Also, we have established that a higher rating means a better interpretation, either because it contains more symbols, or more complex symbols, or more precisely drawn symbols, or a combination of the three. Accordingly, it makes sense to discard all DAG but the one with the greatest rating. If the semantic attribute cannot be evaluated for any root of the DAGs, the sketch in turn cannot be interpreted. An example for attribute evaluation is given in Appendix C.2.

9.7 Summary

This chapter has described how syntax can be checked based on the RHM, and how semantics can be determined based on the derivation structure obtained by parsing. The derivation structure is usually a DAG due to embedding productions. The parser works bottom-up and requires the grammar to be context-free and in CNF. Context-freeness is obtained by the available kinds of production rules, the CNF can be computed automatically. Next to the obvious terminal production rules (TPRs) and nonterminal production rules (NPRs), there are set production rules (SPRs), and embedding production rules (EPRs). SPRs allow for more efficient parsing if a set of edges with the same label is to be derived, and the order of the edges does not matter. EPRs allow for describing rules which are not context-free.

Settlement of conflicts in the RHM is assured for every production by not deducing nonterminals which are derived into conflicting terminals. For set productions, this condition can only be established after the root of the derivation DAG is deduced. Then, a heuristic is used which solves a variant of the clique problem, directed by ratings of edges.

Chapter 10

Evaluation

An obvious question about every approach is its correctness. In some fields it is possible to give a formal proof of correctness. In the case of this thesis, a formal proof cannot be given. It is impossible to show that the approach produces the result intended by the user. Even more, it may happen that this result is not produced at all. Another way to argue about correctness is an empirical evaluation, which is performed in this chapter. It follows two goals. The first is to tell about the actual *recognition rate*, i.e.,

$$\text{recognition rate} := \frac{\text{\# of correctly recognized shapes}}{\text{\# of shapes intended by user}}$$

The higher this rate is, the more often the correct shape is recognized, and the more rarely the user has to manually correct his input. A recognition rate of 100% means that every shape intentionally drawn by the user is correctly recognized. Every approach to sketching strives to get as close to this value as possible, however there always will be misrecognized shapes.

The second goal of the evaluation in this chapter is to assess the performance of the implementation. The faster the actual result is computed, the better the system is likely to be perceived by the user. A long waiting period is bothersome, and should be avoided by implementations.

In order to learn about recognition rate and performance of the implementation, a user study was conducted. 16 participants, all computer scientists or students of computer science, each drew one example sketch of each of the six diagram languages discussed in Chapter 2. For each diagram language a master diagram was shown to the participants, which had to be copied. This assured that the results from the different participants are comparable, as each participant copied the same master. Figure 10.1 shows all six master diagrams. These are equal to the example diagrams from Chapter 2; only for the GUI builder a simpler diagram than the one shown in Figure 2.4 was used.

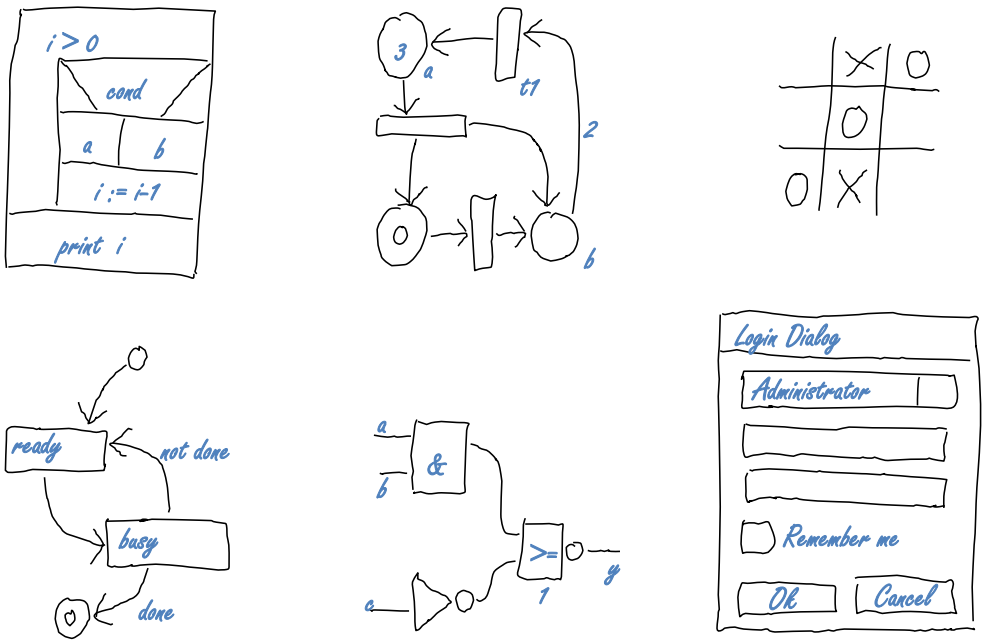


Figure 10.1: The six diagrams used as masters for the user study.

Most of the participants had never or very rarely used a pen display before, so each participant was first allowed a training period to get used to the hardware (a Wacom DTU-710 pen display). For the user study's actual task of sketching, the participants were encouraged to draw freely and unrestricted. To avoid any dependencies from the order in which the six sketches had to be produced, their order was shuffled for each participant. During the whole test, absolutely no feedback was given to the participants about the recognition rate or other system behavior. This means that there was no way to adopt the drawing style to the system. No participant took more than 7 minutes for the training period, and only two participants took more than 5 minutes. The actual sketching was completed in less than 10 minutes for each participant.

From the 16 participants most decided to lay the display flat on the table, which imitates pen and paper more closely. Still, many participants found it hard to place the hand on the display (which is possible), and instead avoided touching the display with their hand. This unnatural hand posture resulted in a diffident drawing style, and was likely to increase imprecision. Regarding the size of the produced sketches, the participants can be arranged into two groups. Participants from the one group used the available screen space very liberal, thus producing larger sketches that were neater. Participants from the other group tended to draw the sketches very small, which had a negative impact both on the recognition rate and processing time.

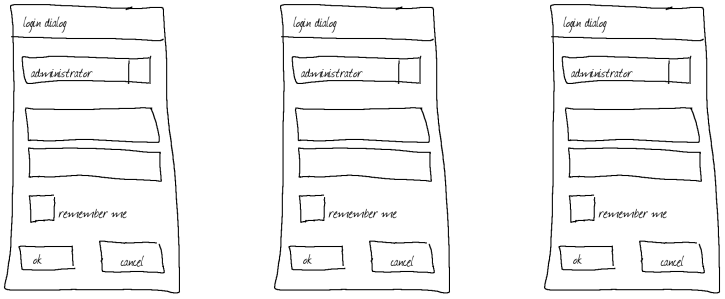


Figure 10.2: A GUI builder sketch from the user study in three copies. The copies are identical and have been placed next to each other automatically in order to linearly increase the load on the system.

The structure of this chapter is as follows. The next two sections discuss processing times (Section 10.1) and recognition rates (Section 10.2) for the $6 \times 16 = 96$ sketches obtained from the user study. The two aspects explained in Section 6.1 and Section 6.3 (elimination of duplicates, and suppression of shapes containing other shapes) have been intended to minimize the load on the analysis, measured as the number of shapes, by removing shapes in the postprocessing step of the recognition stage. Section 10.3 and 10.4 discuss to which extent this goal is achieved, by means of selected examples. Section 10.5 concludes the chapter.

10.1 Processing Time

For each of the 96 sketches obtained by the user study it was measured how long processing took. An ordinary PC was used as hardware[1]. In order to obtain more stable results, each test was repeated ten times, and the average value was taken. Furthermore, to evaluate how the implementation performs when the input size is increased, the complete procedure was repeated nine times for each single sketch, each time with one additional copy of the original sketch to be processed. Each copy was placed with a considerable distance to all other copies to avoid side effects. This way, both the input (counted as number of strokes and number of texts) and the number of recognized shapes grew linearly. One of the participants' sketches of the GUI dialog box with two additional copies is shown in Figure 10.2.

The averaged values for each single sketch were averaged again for all 16 sketches. The results can be seen in Figure 10.3, which shows the mentioned

[1]The PC ran Ubuntu Linux 8.10 64bit (kernel version 2.6.27), and was equipped with an Intel quad core CPU at 2.66GHz and 4GB of physical main memory, and Java SE runtime environment (64bit, build 1.6.0_10-b33). However, only one CPU core and a maximum of 2GB of main memory have been allocated for the tests.

average processing time depending on the number of copies (1 to 10). Standard deviation is shown as vertical black lines for each averaged value. Figure 10.4 shows the fraction of both stages (recognition and analysis) of the total processing time, again broken down by the number of copies.

For all reported values, preprocessing is not included. The reason is that preprocessing is performed incrementally and does not add to the perceived performance of the system. For evaluating the performance of preprocessing single strokes and text, the time for preprocessing was measured for the case of ten copies of each diagram. The average time for preprocessing a single stroke was 0.36ms, the average time for preprocessing text was 0.04ms. These values indicate that the performance impact of preprocessing can indeed be neglected, whether in an incremental setting or not.

What becomes apparent from the actual measured values (not shown) is that most of the time is spent either by the assembler, or by the parser. Accordingly, performance for the assembler and postprocessing are aggregated in Figure 10.4, and so are modeler, reducer, and parser aggregated for the analysis. The maximum aggregated fraction of postprocessing, modeler, and reducer of the total processing time for all sketches for 1 through 10 copies never exceeded 5%. An exception to this rule is made by the GUI builder, where the modeler consumes relatively more time, and the aggregated fraction of postprocessing, modeler and reducer thus increases to about 15%. Still, the aggregation of values as described is justified.

For all six diagram languages the processing time grows roughly linear with the input size. Only for Petri nets (cf. Figure 10.3(a)) and for statecharts (cf. Figure 10.3(d)) a slightly greater increase can be observed. The implemented algorithms are not all in $\mathcal{O}(n)$, but in higher complexity classes. This is especially true for assembler and parser. The assembler, for example, takes considerably more time for more compact sketches, because the input becomes more ambiguous this way. The performance of the parser depends on the number of terminals, and on the production rules. If the rules are not wisely specified, e.g., by not using set productions, complexity increases. See [68] for more details. Despite the over-linear complexity, the mentioned roughly linear time consumption can be observed, which shows that the practical impact of these algorithms is not severe.

From the average total processing time, the average processing time can be computed for a single shape drawn by the user. The result is shown in the following table, dissected by the six diagram languages.

Petri nets	**NSD**	**GUI builder**	**Statecharts**	**BLD**	**Tic-tac-toe**
10ms	23ms	4ms	28ms	3ms	3ms

NSD and statecharts show the highest values. The reason is that the total processing time of these languages is also high, compared to the GUI builder, Tic-tac-toe and BLD. Petri nets also show a low average processing time per shape, although

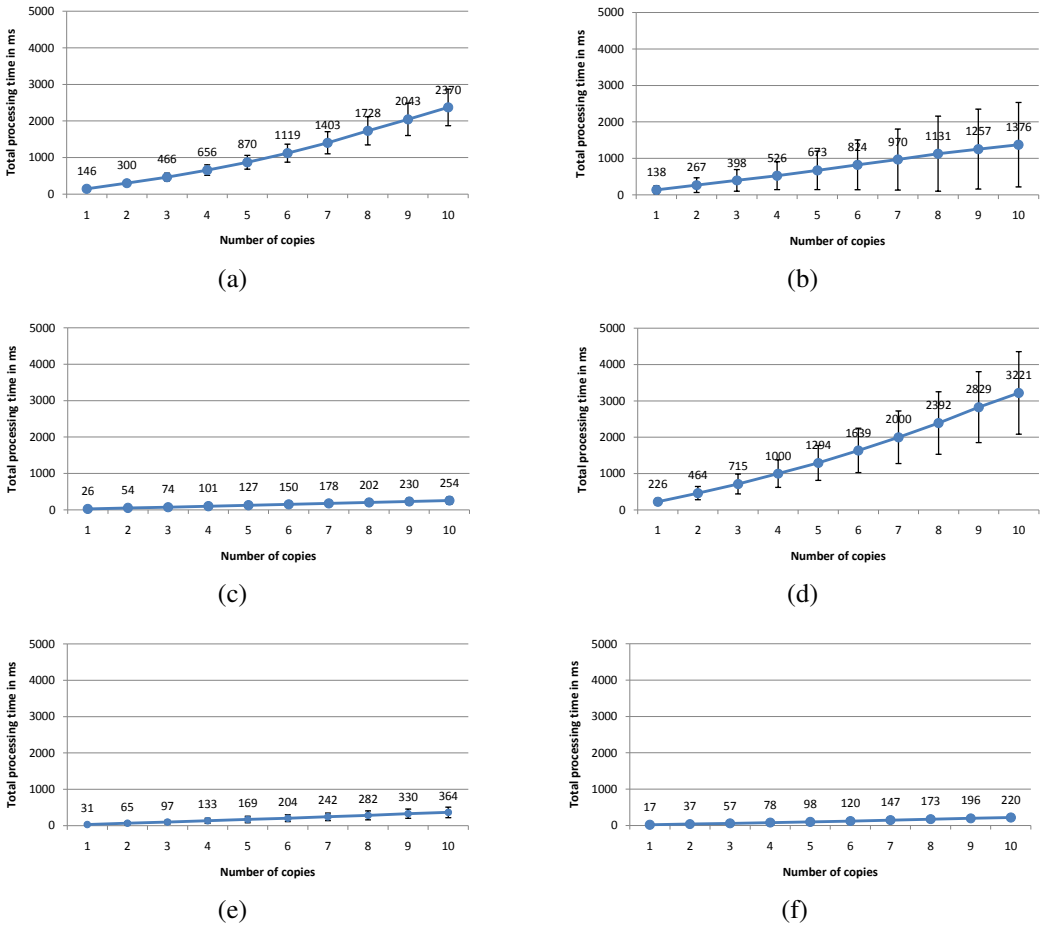


Figure 10.3: Average total processing time for different diagram languages. The vertical black lines show the standard deviation. For (c) and (f) the standard deviation is too small to be shown. (a) Petri nets. (b) NSDs. (c) GUI builder. (d) statecharts. (e) BLDs. (f) Tic-tac-toe.

they take the second most time for processing. However, the Petri net master comprises 13 shapes, compared to 9 shapes for statecharts, and only 6 shapes for NSD.

For NSD, as well as for statecharts, the recognition stage contributes much more to the total processing time than the analysis stage does (cf. Figure 10.4). NSD shows a disadvantageous visual appearance of its shapes, which lead to the recognition of too much shapes. This issue has been discussed before and led to the addition of the configuration option to remove these false shapes (cf. Section 6.3). However, these shapes are still recognized, so the option cannot improve the performance of the recognition stage, but only for the analysis stage (cf. Sec-

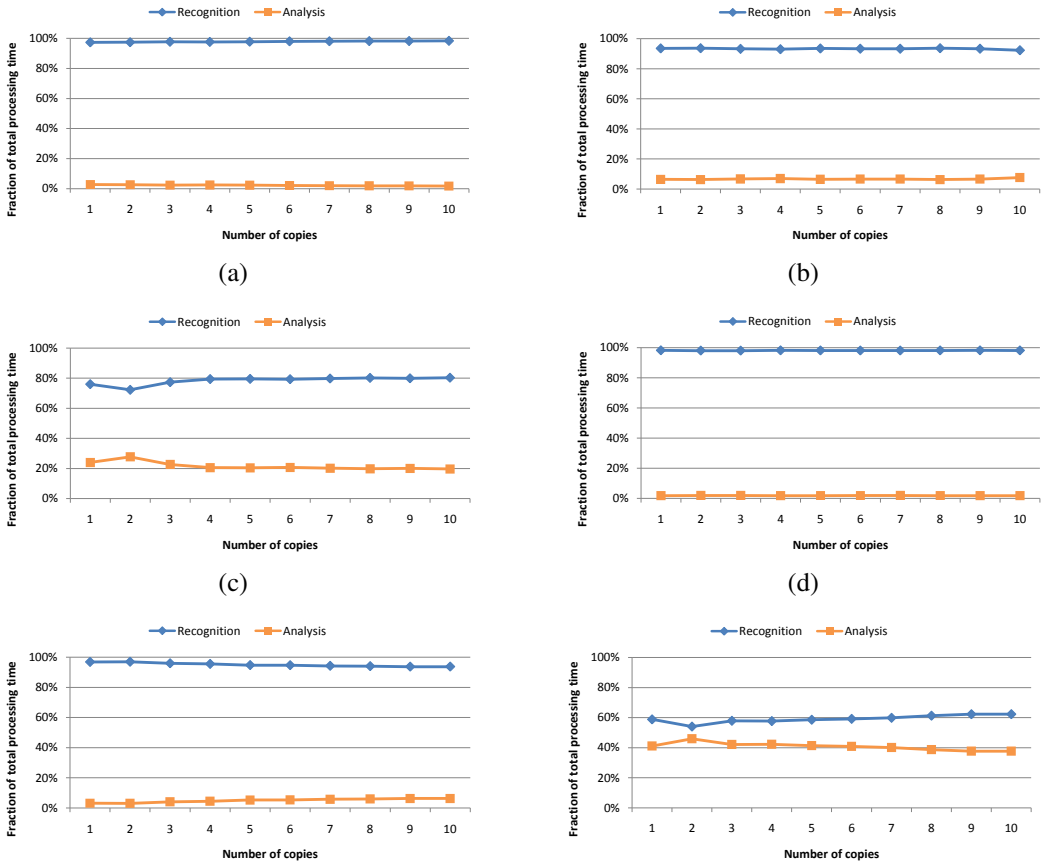


Figure 10.4: Average fraction of recognition and analysis stages of total processing time. (a) Petri nets. (b) NSD. (c) GUI builder. (d) statecharts. (e) BLD. (f) Tic-tac-toe.

tion 10.4). For statecharts, it could be observed that the two circles forming a final state led to many possible combinations of arcs by the assembler. Of course, most of these combinations were false, and thus discarded, but they still have an impact on the processing time.

Common to all six diagram languages is that the fractions of recognition and analysis are independent of the input size (cf. Figure 10.4). The difference lies only in the actual values. In general, the recognition stage consumes more time than the analysis stage, even more than 90% of the total processing time for four of the six diagram languages. For the GUI builder and for Tic-tac-toe, the analysis stage has a higher fraction of the total processing time. However, in these two cases the recognition stage performed very well, and the total processing time is the lowest.

10.2 Recognition Rates

As mentioned in the introduction to this chapter, the other relevant aspect next to processing time are the recognition rates of the implementation. To begin with, these are some terms we use in the following.

- a *true positive* is a shape that the user has drawn intentionally, and that is correctly recognized. For example, each of the three transitions contained in the Petri net master (cf. Figure 10.1) has been drawn intentionally.
- a *false positive* is a shape that is recognized, but which has not been intended by the user. False positives may happen due to tolerances applied by the recognizer, or due to misleading diagram syntax (for example, NSD, cf. Figure 6.2).
- a *false negative* is a shape that the user has drawn intentionally, but which is missed by the recognizer, for example, because it has been drawn too sloppily.

Using the 96 sketches from the participants, the number of true positives is divided by the number of intended shapes (as given by the master of each of the six diagram languages) in order to obtain the recognition rate for a single sketch. These recognition rates are averaged for the six diagram languages. Figure 10.5 shows the result. Except for Petri nets and statecharts, recognition rates range around 90%. For the two exceptions, recognition rates are lower because the user study's participants tended to draw arrows (which occur only in these two diagram languages among the six) sloppily, which led to many false negatives. Also, rectangles, either used to express transitions, or to express states, were drawn considerably more imprecise.

For the final result, obtained by the analysis stage, the following observations hold for *each* of the 96 sketches.

- Either the result was as intended by the user, i.e., it contained only true positives and no false negatives,
- or the result contained only intended shapes (true positives), but was incomplete due to shapes missed by the recognizer (false negatives),
- or false positives were included in the result in order to obtain a syntactically correct result at all, which would not have been possible with the true positives only.

The third item is very important. It means that, in no case, a false positive was included in the final result of processing in favor of a true positive. This fact

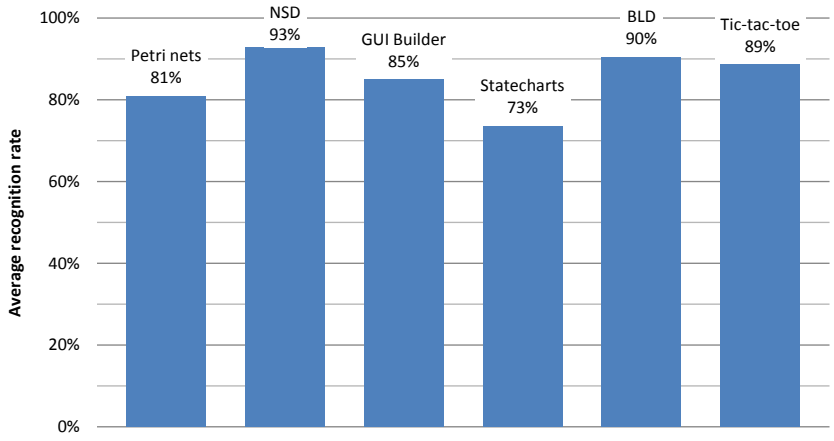


Figure 10.5: Average recognition rates for the 16 diagrams for each of the six examples.

is an important indicator to show that the analysis stage is correct and produces valid results. For example, for Petri nets this means that tokens and places were always correctly told apart. For NSD this means that no false positives similar to those suggested by Figure 6.2 were included in the result, given that the smaller, overlapped shapes were also recognized.

Note that the recognition rates shown in Figure 10.5 are calculated based on the results of the recognition stage. It makes no sense to use the results of the analysis stage for this purpose, since this stage produces an empty result if no syntactically and semantically correct diagram can be found. The correctness of the diagram relies in some cases on one single shape. For example, each statechart must have an initial state. Each GUI builder sketch must contain a window containing the single controls. Sketches of Tic-tac-toe require a grid. In these cases, if the one crucial shape is missed by the recognizer, the result becomes empty, no matter how many intended shapes were recognized.

10.3 Effect of the Elimination of Duplicates

The procedure described in Section 6.1 removes duplicates of shapes, which occur due to the tolerances applied by the assembler. The idea is to spend a comparatively small amount of processing time on the removal process, in order to save much more time in the analysis stage. This section examines the effect of this procedure with the implementation.

Figure 10.6 shows the average total processing times for the six example languages with and without elimination of duplicates, for one copy of the diagram.

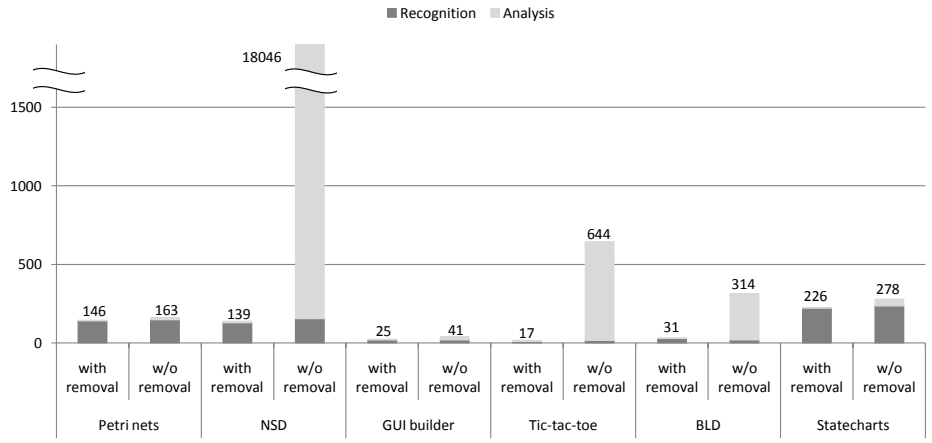


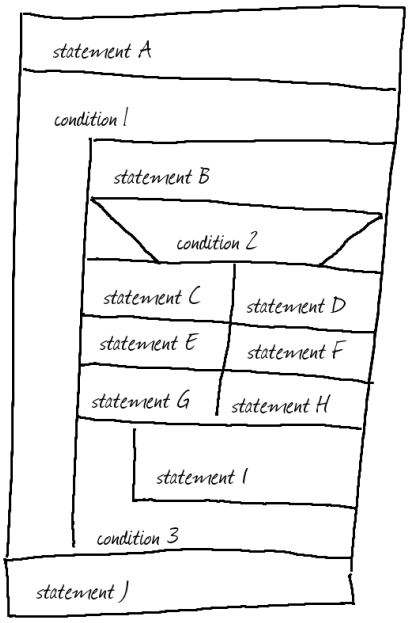
Figure 10.6: Average total processing time with and without removal of duplicates for one copy of the diagrams, stacked by time for recognition and analysis.

The bars are stacked by time for recognition and analysis. As expected, recognition time (which includes postprocessing where duplicates are removed) remains nearly constant, while analysis time increases by not removing duplicates. Common to all diagrams is that the recognition rates reported in the previous section were not affected by removal of duplicates.

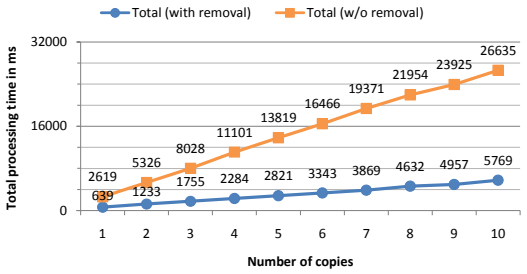
For Petri nets, for statecharts, and for the GUI builder, removal of duplicates only shows a small impact. For NSD, for Tic-tac-toe, and for BLD, analysis time increases considerably if duplicates are not removed. In the case of NSD the average total processing time even grows to about 18 seconds. The problem of NSD is that some of the 16 sketches show a high number of duplicates. If these duplicates are not removed, the imposed load on the analysis is significant. If duplicates were not removed, for the three sketches with the highest number of recognized shapes which were passed to analysis, the average number of these shapes was 147. In contrast, if duplicates were removed, the average number of shapes for analysis of the same three sketches is only 10. Accordingly, the speed-up for NSD achieved by removal of duplicates is very high. For Tic-tac-toe, and for BLD, the speed-up is not as high as for NSD, but still remarkably.

10.4 Effect of the Suppression of Shapes Containing Other Shapes

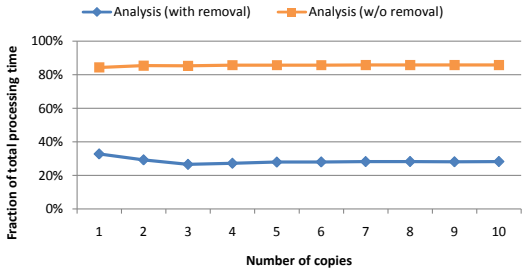
Section 6.3 has explained the rationale behind the removal of shapes containing other shapes. Now it is investigated whether these considerations hold for the implementation. If the 16 NSD sketches are processed again, without shapes con-



(a)



(b)



(c)

Figure 10.7: Processing time for NSD with and without removal of shapes containing other shapes. (a) a sketched NSD. (b) average total processing time for the sketch for linear increase in load. (c) average fraction of the analysis stage of the total processing time.

taining other shapes being removed, a negative effect is observed: the average total processing time roughly doubles. The situation becomes even more severe if another example is used. The example NSD diagram for the user study is very small and has a simple structure (cf. Figure 10.1). Figure 10.7(a) shows a more complex NSD. It comprises 13 shapes. Using this sketch, the positive effect of the removal of shapes containing other shapes is greater.

For this sketch, the recognition stage yielded 29 shapes if shapes containing other shapes were removed, otherwise 101 shapes. Accordingly, the analysis stage produced different numbers of possible solutions (derivation DAGs where the semantic attribute could be evaluated, cf. Section 9.6), 1154 in the case of removal, otherwise 20964. Obviously, the processing time has to differ in a similar relation. Figure 10.7(b) shows the average total processing time with removal, and without removal of shapes containing other shapes, again for 1 to 10 copies of the sketch. While the time for the recognition stage is invariant to the setting (removal of shapes containing other shapes took virtually no time), all the extra time consumed if no shapes were removed was consumed by the analysis step. Figure 10.7(c) confirms this fact. If shapes containing other shapes were not re-

moved, the fraction of the analysis stage on the total processing time increased from about 30% to about 85%.

Like in the section before, the recognition rate is not affected by the option. Consequently, the assumption made for the removal of shapes containing other shapes is valid in this test setting.

10.5 Conclusion

In this chapter, performance and recognition rates of the implementation are evaluated based on a user study. The overall result of the evaluation is that the approach is fast, with average processing times per shape ranging from 3ms to 28ms, and that the recognition rates are good, with about 90% for four of the six diagram languages. The measures arranged to improve processing time (elimination of duplicates, suppression of shapes containing other shapes) all work as intended. On the one hand, they greatly improve performance, but on the other hand they do not influence recognition rates.

Taking a closer look at the evaluation results reveals interesting intricacies. Both the performance and the recognition rates strongly depend on the diagram language. This dependency is an important aspect, because the approach presented in this thesis is generic, and has been designed to handle different diagram languages. However, the six example languages were carefully chosen in order to be representative for a broad range of diagram languages (cf. Chapter 2).

Sketches where processing is not as fast (worse than the average), or recognition rates are not as good (worse than the average), nearly always exhibit a certain drawing style. The style can be characterized by the sketch size and the precision. There is a clear tendency that the less space a sketch covers on the canvas, the higher its processing time is. This is due to the tolerances used in the recognition stage. All respective threshold values are generously set in order to allow for an easy and unconstrained drawing, and remain fixed for each participant of the user study. If sketches are drawn very small, the threshold values are too high, resulting in much more combinations of primitives, and ultimately in more shapes that are recognized.

Likewise, the less precision that is applied by the user, the lower the recognition rate is. A typical behavior is to draw corners, e.g., from a rectangle, more rounded than angular. Similar, most users drew the open-head arrows from Petri nets and statecharts by first drawing the shaft in one stroke, and then drawing the arrow head in a second stroke. Here, a similar observation could be made: the arrow head was sometimes drawn like an arc, and not angled. On the other hand, drawing straight lines, e.g., for transitions of Petri nets, or for NSD, or for BLD, was rarely a problem. Circles, used in Petri nets, Tic-tac-toe, BLD, and

statecharts, were always drawn in one stroke, without any exception. Still, the individual recognition rates for circles differ noticeably between the four diagram languages. For Petri nets (92%) and Tic-tac-toe (95%) the recognition rates are good, while for statecharts (79%) and BLD (72%) the values are lower. It could be observed that circles were often drawn quite small in the sketches of these two diagram languages.

Two final aspects have to be considered in order to assess the recognition rates reported in this chapter. The first is that the approach recognizes individual shapes from complete sketches. Segmentation and clustering are performed automatically by the assembler step, based on the model data. There is no other indication given to decide about these issues. This way, the drawing process is more comfortable and natural for the user, but recognition becomes more difficult. The other aspect influencing the recognition rate is that the participants were not shown any feedback about the recognition before they were finished drawing. Hence, the individual drawing style could not be adapted to the system. Without this stringent precondition, higher recognition rates can be expected. The aforementioned issues of small sketches, sloppy corners and sloppy arrow heads would be reduced. Despite these two aspects, the implementation accomplished its task well, both in terms of performance and recognition rate.

Future work can be derived from the conclusion. The mentioned issues of imprecise arrows and corners of rectangles could be dealt with by more sophisticated transformers, which take these issues into account. Then, recognition rates would certainly improve. For example, the line transformer could elongate two perpendicular lines in order to find their point of intersection as the corner point that is missed due to the sloppy drawing style.

A more sophisticated approach to sloppy drawing style could be the introduction of top-down aspects. The complete sketch processing chain described in this thesis works bottom-up. If there are (sub-)strokes that do not contribute to any shape, the system ignores these (sub-)strokes. But if we assume that the user intends all of his strokes to represent something meaningful, these (sub-)strokes should not be discarded. Accordingly, after the assembler has finished, all unused (sub-)strokes (and maybe other strokes in their direct spatial neighborhood) could be extracted and searched again for shapes, but with relaxed thresholds. The ratings of the additional shapes that are recognized this way could be reduced in order to make the analysis stage prefer those shapes which have been found in the first run of the assembler where the regular thresholds are applied.

Also an open question is how DSKETCH is perceived by users over a longer period of time. Long-term observations of users could help to improve both DSKETCH and its user interface in order to meet expectations and requirements of users. Thereby, also the question could be answered of how much recognition rates improve if the user is used to the system and has learned about its behavior.

Chapter 11

Summary and Conclusion

This thesis has explained DSKETCH, a generic approach to the understanding of hand-drawn diagrams. Using specifications it can be tailored to specific diagram languages. Understanding a diagram comprises two stages, recognition and analysis. The recognition stage identifies the shapes drawn by the user. The analysis stage then analyzes the shapes in their context to resolve ambiguities. This process is based on syntax and semantics of the diagram language. Each of the two stages is made of three subsequent steps.

The first step of the recognition is preprocessing of strokes and text, which is performed by independent transformers. Each transformer has a certain view and processes data accordingly. For example, the line transformer maps the strokes to straight lines, but does not regard other primitives. Thereby data gets abstracted, which simplifies later processing, and decreases the amount of data. Each transformer creates one model where the preprocessing result is stored. The advantage of this procedure is that there are several models in parallel. As all strokes are processed by each transformer, several interpretations of a stroke may coexist in different models. There is no need at this point to decide for one interpretation (which is not even possible yet, as there is no information to guide this decision). It is completely up to the transformers what preprocessing they perform. For example, a transformer can try and map the input data to one kind of primitive, as the line transformer does. A transformer could also map the input data to different kinds of primitives, although no transformer in DSKETCH does so. Finally, there can even be more than one transformer for one kind of primitive, as it is the case for the arc transformer and the circle transformer.

The assembler is the second step in the recognition stage. Based on the data in the models, it identifies shapes. The original input data, i.e., strokes and text, are not present to the assembler. The rationale behind this is that the models only contain highly abstracted data. Consequently, the assembler computes its result very efficiently if it looks at the models only, and not at the verbose input data.

This means that all relevant data must be processed by transformers, otherwise it is lost. The assembler identifies shapes by querying primitives from the models, and combining these primitives. A search plan, automatically generated from the specification of shapes, guides this process and defines the order in which primitives are searched for. This allows for searching primitives shared by different shapes first, which makes the whole process more efficient. For example, places and tokens from Petri nets, both drawn as circles, are identified in parallel and do not have to be searched for a second time. An important aspect of the assembler is that it has no specific information on any model. If a primitive is searched for, every model is queried, and the results from all models are used to continue the search process. This is a crucial aspect to the extensibility of the recognition stage. New transformers and assigned models can easily be added, and the assembler does not have to be changed in any way.

The third step of the recognition stage is the postprocessing. The set of shapes identified by the assembler is first examined for duplicates. Due to the tolerances applied by the transformers and the assembler, it frequently happens that the same shape is identified more than once, each time slightly different. These duplicates are correctly handled by the analysis stage, i.e., this stage never includes duplicates in the final result. However, removal of duplicates during postprocessing improves the performance of the analysis stage. Using a further configuration option, set in the specification, the postprocessing step can remove even more shapes to increase performance. This option, motivated by NSD, allows for suppression of shapes containing other shapes. It must be optional, because its use depends on the shape specifications and thus it cannot be applied in general. Moreover, the postprocessing identifies ambiguities between shapes. An ambiguity means that only one of two shapes can be correct. Ambiguities are explicitly modeled as so-called conflicts for the analysis stage, which has shown to be a valuable approach to their resolution. Finally, all shapes that have not been removed are rated. The rating is later used to decide for one of two shapes, or one of two sets of shapes, in case that the context information does not suffice to make this decision.

With the postprocessing step the recognition stage is over. The subsequent analysis stage computes a syntactically and semantically correct subset of the recognized shapes, and respects the conflicts identified earlier. For this purpose, the complete stage is based on the DIAGEN tool, and adopts three of its steps. The first step is the modeler. It identifies relations between shapes and creates a hypergraph model representing all shapes and all relations. The specification describes attachment areas defining which parts of shapes can be related to parts of other shapes at all. The modeler takes into consideration that these attachment areas can be deformed for actual shapes, due to the impreciseness of hand-drawing.

The second step of the analysis stage is the reducer, which reduces the hypergraph model produced by the modeler in order to yield another graph model,

the reduced hypergraph model. This reduction process is guided by rules. The key idea behind the reducer is to decrease the size of the hypergraph model by replacing patterns with simpler patterns, comprised of less edges and nodes. In doing so, invalid patterns are ignored, i.e., not reduced, and thus need not be dealt with later. An example for an invalid pattern is an arrow in a Petri net connecting two places. During the reduction process it is assured that conflicts between the shapes are still present in the reduced model. Furthermore, negative application conditions are a second source of conflicts. These special conditions are used to constrain the application of reduction rules. In our case, the challenge is that rule applications must not be omitted if these conditions are satisfied by misrecognized shapes. As a solution, rule applications are never omitted, but conflicts are generated for matched application conditions as well. An advantage of DSKETCH is that conflicts between shapes and conflicts due to these application conditions are represented by the same concept in the reduced model, which simplifies the subsequent processing step.

The final step in the analysis stage, and in understanding the complete sketch therewith, is the parser. Controlled by production rules defined in the specification, it creates a derivation structure in a bottom-up manner. The edges in the reduced hypergraph model are used as terminal symbols. The production rules are essentially context-free. However, there are embedding production rules which allow for the specification of not context-free language constructs. Furthermore, there are set production rules which can be used to significantly improve the performance of the parsing process. For the application of each production rule, the parser respects conflicts by not using two terminal or nonterminal symbols in the RHS of an application of a rule where the symbols are conflicting. The result of the parsing process is a derivation DAG, or a set of DAGs. In the latter case, that DAG is chosen which exhibits the highest rating. Then, semantics of the sketch are computed on the basis of this DAG. The semantics is defined as the value of one designated attribute of the start symbol. The value of this attribute is the final result of sketch processing.

The complete approach has been thoroughly evaluated, based on a user study. 16 participants each drew six sketches from different domains. Performance and recognition rates have been evaluated using a prototypical implementation. This evaluation has shown that both are satisfying. A general rule of thumb cannot be given on how to estimate both measures, as they depend on the diagram language, but also on the actual sketch. The evaluation has also shown that the assumptions underlying the removal of duplicates and the suppression of shapes containing other shapes applied by the postprocessor (to reduce the number of shapes) are valid, because their use improves processing time considerably, and does not degrade recognition rates.

The two central aspects DSKETCH is designed for are the understanding of

sketched diagrams, while preserving a natural and flexible way of drawing for the user. The approach combines the following attributes, which make it unique and illustrate the scientific relevance.

- Generic – the approach is not customized to a specific diagram language, but flexible. Using specifications, tailoring can be achieved. On top of this, also the final result of diagram processing can be customized to domain-specific needs.

- Two stages – diagram processing is comprised of the recognition stage and the analysis stage. The recognition stage relies on transformers and models, and is very flexible and extensible. The analysis stage is based on the powerful DIAGEN framework, and performs diagram understanding by creating the final processing result.

- Multi-stroke recognition – an important benefit of sketching is its natural way of input. This benefit should not be mitigated by constraining the user's drawing style.

- On-line recognition – based on the motivation that sketching editors replace traditional point-and-click editors, on-line stroke information is present and can be exploited.

- Automatic clustering and segmentation – diagrams comprise more than one shape. Accordingly, DSKETCH recognizes shapes in completed sketches, and performs all necessary stroke clustering and segmentation automatically.

- Geometry-based specification and recognition of shapes – like in other approaches, shapes are specified by primitives and their geometric relations.

- No limitations by features – the architecture of the recognizer does not rely on features in general. A drawback of features is that they restrict the user's drawing style in order to match the features, if they are not chosen wisely. Our only application of features is for the circle transformer.

- No training – as shapes are specified, and as recognition is not based on features (apart from the circle transformer) no training samples are required, which decreases start-up time for the user.

- Automatic resolution of ambiguities – ambiguities are solved automatically, not based on inflexible meta-rules, but based on diagram language-specific rules describing syntax and semantics.

- Lazy recognition – because only the user knows when a diagram is finished, and because correct syntax and semantics are crucial, it is the user who has to explicitly invoke recognition.

- Checking of syntax and semantics – automatic resolution of ambiguities is interwoven with checking syntax and semantics. Using this checking it is possible to compute a processing result that can be used immediately in another application, which is a typical scenario for sketching.

- Good performance and recognition rates – shown by a user study, performance and recognition rates are good.

The presented thesis has identified several starting points for further work, which have been illustrated in their context in the respective chapters. For each such chapter, a single section has been devoted exclusively to further work. Accordingly, we give only a brief overview here, without delving into all the details that have been discussed before.

- For the preprocessing step, the use of a divider has been discussed, which would allow for mode-less writing and drawing next to each other on the canvas (Section 4.6).

- Coupled with the issue of a divider is to evaluate which text input metaphor is more convenient to the user, different alternatives are conceivable (Section 4.6).

- More kinds of primitives could be useful, for example, hatched and filled regions, or dashed and dotted lines (Section 4.7). Also, repetitive patterns like spirals or helices can neither be specified nor recognized.

- Low-level recognizers from other research groups can be integrated into DSKETCH as an independent transformer, which could increase overall recognition rates (Section 4.7).

- The evaluation revealed further issues of sloppy drawn arrows and corners that could be handled during the preprocessing step, or even better by a top-down approach (Section 10.5).

- The assembler, and all subsequent processing steps, could benefit from an incremental approach, as this can be expected to greatly increase performance. Other approaches that do have incremental processing are usually very efficient, which supports this assumption (Section 5.6).

- As an alternative to an incremental approach, or in addition, recognition could be parallelized, as all search states involved in recognition are independent of each other (Section 5.6).

- For the reducer, this thesis suggests a different mechanism to apply negative application conditions. This mechanism may also be useful for respective diagramming tools which do not rely on sketching, as it may help to give the user a better feedback in the case of an erroneous diagram (Section 8.4).

A more general issue for further work, not discussed so far, regards the user interface of our sketching application. The focus of this thesis only lies in the technical aspects of sketching. However, a mature user interface that supports the user in his work is crucial to the overall success of sketching. This includes, for example, the question which typical workflows occur in a sketching application, and what the user interface provides in order to enable such workflows in a flexible and natural manner. This issue is tightly coupled with the research field of user interfaces for pen-based computers and HCI.

Appendix A

Specification of Diagram Languages

This chapter shows the complete specifications for all six examples of diagram languages illustrated in Chapter 2. For each diagram language is shown the specification of

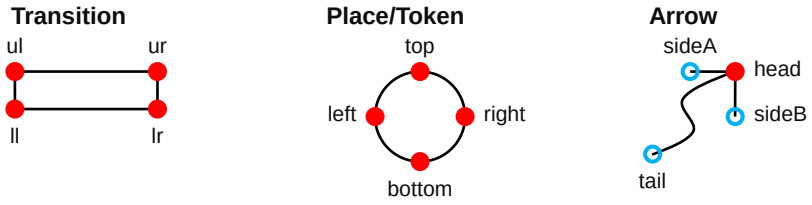
- The shapes, comprising primitives (cf. Section 1.2), constraints (cf. Section 5.1), and attachments areas (cf. Section 7.1).
- The settings for the configuration options applicable in the postprocessing step (cf. Sections 6.2 and 6.3).
- The relation types (cf. Section 7.2).
- The terminal and nonterminal symbols.
- The reduction rules (cf. Section 8.2).
- The production rules (cf. Sections 9.1 through 9.4), and rules for attribute evaluation (cf. Section 9.6).

A graphical representation of the specification is chosen where appropriate. For text primitives, there is sometimes given a regular expression which further describes the text. For the specification of constraints there is a choice between more constraints, which lead to a more precise shape, and fewer constraints, which allow more freedom in drawing shapes. We tend to use less constraints in order to not over-constrain drawing.

The shape represented by a shape edge e is denoted as e.self(). For attribute evaluation there are often function calls used, which are not further explained. However, meaningful names are chosen to assure understandability. By convention, labels of all edges representing terminal symbols start with t_, while labels of all edges representing nonterminal symbols start with n_. The numbers used to distinguish the nodes visited by an edge are omitted if their mapping is obvious.

A.1 Petri Nets

This section shows the specification for Petri nets (cf. Section 2.1). The final result of sketch processing is a model of the sketched Petri net, if such a model can be found. Note that the examples given in Chapters 7 through 9 sometimes differ from the specification in this section, which has been necessary to better describe aspects of the approach which would otherwise not become obvious.



A.1.1 Shape *Place*

primitives

arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right
text within polygon top_left-top_right-bottom_right-bottom_left, optional,
regex `\d+`, unique id capacity
text at polyline top_left-top_right-bottom_right-bottom_left, optional,
regex `[a-zA-Z_].*`, unique id label

attachment areas

polyline top_left-top_right-bottom_right-bottom_left, label place
polygon top_left-top_right-bottom_right-bottom_left, label place_for_tok

A.1.2 Shape *Transition*

primitives

line from ul to ur, horizontal right, unique id line1
line from ll to ul, vertical up, unique id line2
line from lr to ll, horizontal left, unique id line3
line from ur to lr, vertical down, unique id line4
text at polyline line1-line2-line3-line4, optional,
regex `[a-zA-Z_].*`, unique id label

attachment areas
polyline line1-line2-line3-line4, label trans

A.1.3 Shape *Token*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right

attachment areas
polygon top_left-top_right-bottom_right-bottom_left, label token

A.1.4 Shape *Arrow*

primitives
line from head to sideA, arbitrary direction, unique id lineA
line from head to sideB, arbitrary direction, unique id lineB
link from head to tail, unique id shaft
text at polyline shaft, optional, regex `\d+`, unique id cost

constraints
distance head-tail greater than 80, soft constraint
distance head-sideA greater than 50, soft constraint
distance head-sideB greater than 50, soft constraint
distance head-sideA less than distance head-tail, hard constraint
distance head-sideB less than distance head-tail, hard constraint
angle sideA-head-tail greater than 20°, hard constraint
angle sideA-head-tail less than 70°, hard constraint
angle tail-head-sideB greater than 20°, hard constraint
angle tail-head-sideB less than 70°, hard constraint

attachment areas
point head, label arrowEnd
point tail, label arrowEnd

A.1.5 Postprocessing

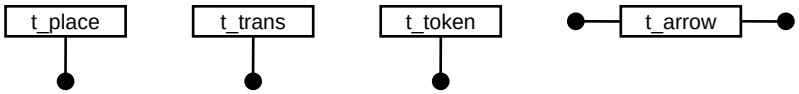
identify conflicts := true
remove larger shapes := false

A.1.6 Relation Types

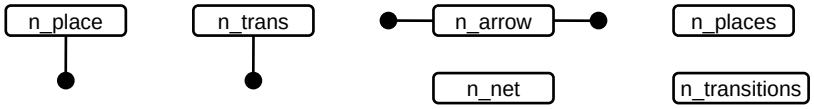
from place_for_tok to token, rigid, label has_token
from arrowEnd to trans, not rigid, label at_trans
from arrowEnd to place, not rigid, label at_place

A.1.7 Parser Symbols

terminal symbols



nonterminal symbols (start symbol is n_net)



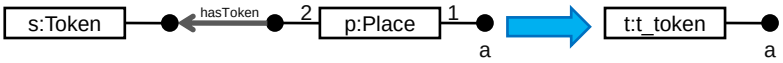
attributes of the terminal symbols
t_place.shape for the original shape that has been recognized
t_trans.shape for the original shape that has been recognized
t_arrow.shape for the original shape that has been recognized

attributes of the nonterminal symbols
n_place.model for the place that is created for the final result
n_trans.model for the transition that is created for the final result
n_places.set for the set of all places
n_transitions.set for the set of all transitions
n_net.model for the Petri net that is created for the final result, semantic attribute
n_arrow.model for the arrow that is created for the final result

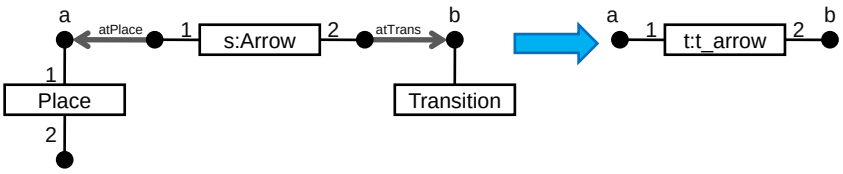
A.1.8 Reduction Rules

t.shape := s.self()

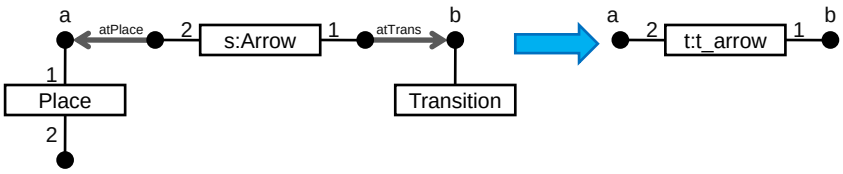
t.shape := s.self()



Condition s.self().radius < p.self().radius / 2

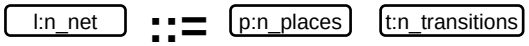


t.shape := s.self()

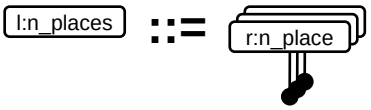


t.shape := s.self()

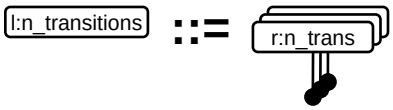
A.1.9 Production Rules



l.net := createNet(p.set, t.set)



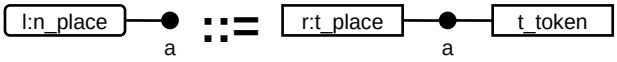
l.set := takeAll(r.model)



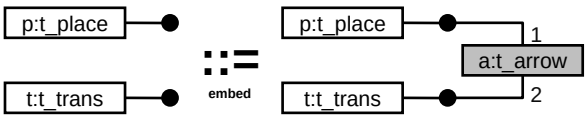
l.set := takeAll(r.model)

l.model := createPlace(r.shape)

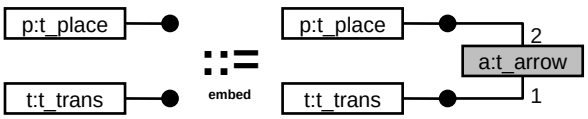
l.model := createPlaceWithToken(r.shape)



l.model := createTransition(r.shape)



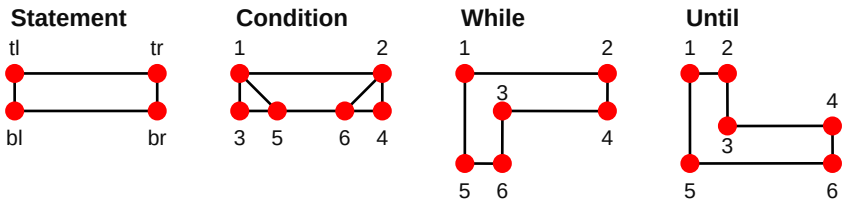
addArrow(t.shape, a.shape, p.shape)



addArrow(p.shape, a.shape, t.shape)

A.2 Nassi-Shneiderman Diagrams

This section shows the specification for NSDs, which are illustrated in Section 2.2. The final result of sketch processing is the pseudocode represented by the sketch.



A.2.1 Shape *Statement*

primitives
line from tl to tr, horizontal right, unique id line1
line from bl to tl, vertical up, unique id line2
line from br to bl, horizontal left, unique id line3
line from tr to br, vertical down, unique id line4
text within polygon line1-line2-line3-line4, unique id statement

attachment areas
point tl, label corner
point tr, label corner
point bl, label corner
point br, label corner

A.2.2 Shape *Condition*

primitives
line from 1 to 2, horizontal right, unique id top
line from 5 to 6, horizontal right, unique id bottom
line from 1 to 3, vertical down, unique id left
line from 2 to 4, vertical down, unique id right
line from 1 to 5, diagonal down right, unique id diag_l
line from 2 to 6, diagonal down left, unique id diag_r
line from 3 to 5, horizontal right, unique id small_l
line from 6 to 4, horizontal right, unique id small_r
text within polygon top-diag_r-bottom-diag_l, unique id condition

attachment areas
point 1, label corner
point 2, label corner
point 3, label corner
point 4, label corner

A.2.3 Shape *While*

primitives
line from 1 to 2, horizontal right, unique id top
line from 1 to 5, vertical down, unique id left
line from 5 to 6, horizontal right, unique id bottom
line from 2 to 4, vertical down, unique id right
line from 3 to 4, horizontal right, unique id hori
line from 3 to 6, vertical down, unique id vert
text within closed polygon top-right-hori, unique id condition

attachment areas
point 1, label corner
point 2, label corner
point 3, label corner
point 4, label corner
point 5, label corner
point 6, label corner

A.2.4 Shape *Until*

primitives
line from 1 to 2, horizontal right, unique id top
line from 1 to 5, vertical down, unique id left
line from 5 to 6, horizontal right, unique id bottom
line from 2 to 3, vertical down, unique id vert
line from 3 to 4, horizontal right, unique id hori
line from 4 to 6, vertical down, unique id right
text within closed polygon hori-right-bottom, unique id condition

attachment areas
point 1, label corner
point 2, label corner
point 3, label corner
point 4, label corner

point 5, label corner
point 6, label corner

A.2.5 Postprocessing

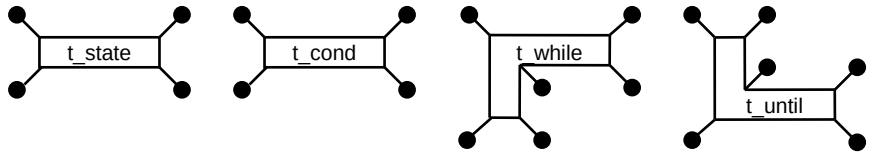
identify conflicts := false
remove larger shapes := true

A.2.6 Relation Types

from corner to corner, not rigid, label connect

A.2.7 Parser Symbols

terminal symbols

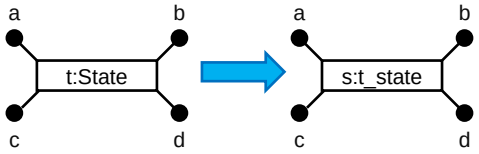


nonterminal symbols (start symbol is n_nsd)

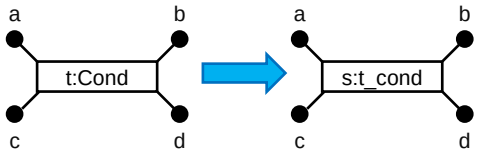
attributes of the terminal symbols
t_state.statement for the statement represented by the edge
t_cond.condition for the condition of the statement
t_while.condition for the condition of the statement
t_until.condition for the condition of the statement

attributes of the nonterminal symbols
n_state.code for the pseudocode represented by this symbol
n_nsd.code for the pseudocode represented by this symbol, semantic attribute

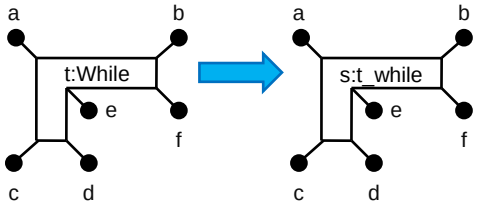
A.2.8 Reduction Rules



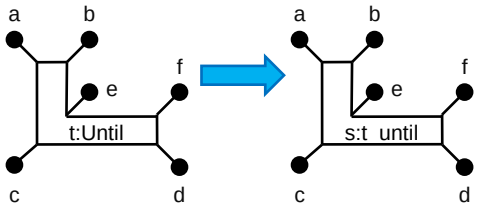
t.statement := s.statement



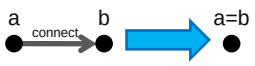
t.condition := s.condition



t.condition := s.condition

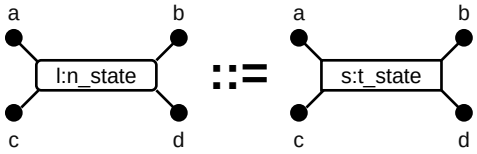


t.condition := s.condition

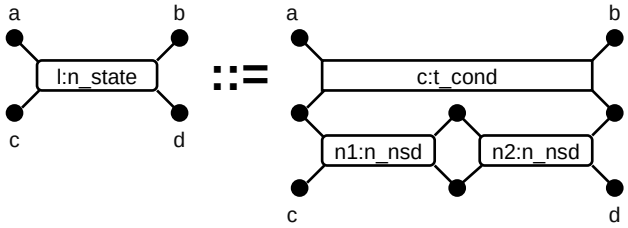


note: this reduction rule merges nodes a and b in the RHM.

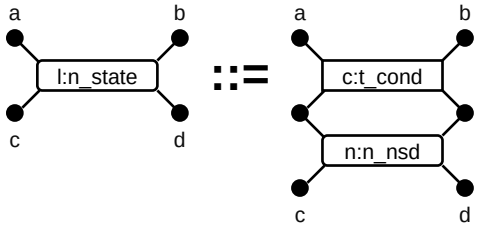
A.2.9 Production Rules



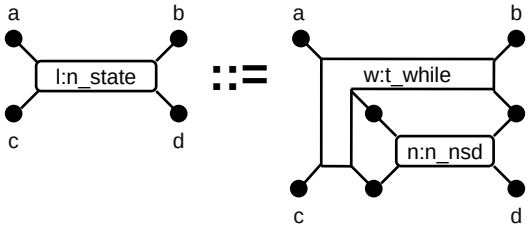
l.code := s.statement



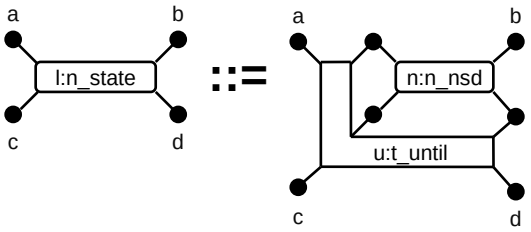
l.code := ”IF ” + c.condition +
” THEN ” + n1.code + ” ELSE ” + n2.code + ” FI”



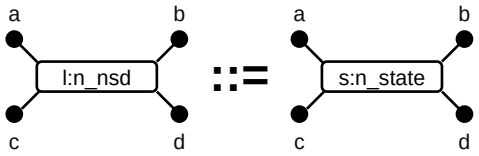
l.code := ”IF ” + c.condition + ” THEN” + n.code + ” FI”



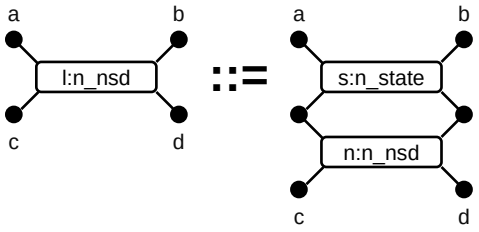
l.code := ”WHILE ” + w.condition + ” DO ” + n.code + ” OD”



l.code := ”DO ” + n.code + ” UNTIL ” + u.condition



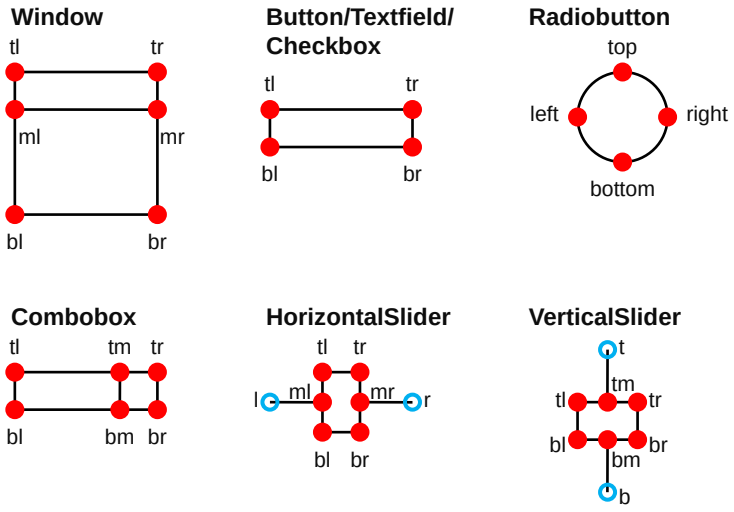
l.code := s.code



l.code := s.code + ”; ” + n.code

A.3 GUI Builder

This section shows the specification for the GUI builder, which is illustrated in Section 2.3. The final result of sketch processing is an actual dialog containing all sketched control elements.



A.3.1 Shape *Window*

primitives
line from tl to tr, horizontal right, unique id top
line from ml to mr, horizontal right, unique id middle
line from bl to br, horizontal right, unique id bottom
line from tl to ml, vertical down, unique id top_left
line from ml to bl, vertical down, unique id bottom_left
line from tr to mr, vertical down, unique id top_right
line from mr to br, vertical down, unique id bottom_right
text within polygon top-top_right-middle-top_left, unique id caption

attachment areas
polygon middle-bottom_right-bottom-bottom_left, label container

A.3.2 Shape *Button*

primitives
line from tl to tr, horizontal right, unique id top
line from bl to br, horizontal right, unique id bottom
line from tl to bl, vertical down, unique id left

line from tr to br, vertical down, unique id right
text within polygon top-left-bottom-right, unique id caption

attachment areas
point tl, label element

A.3.3 Shape *Textfield*

primitives
line from tl to tr, horizontal right, unique id top
line from bl to br, horizontal right, unique id bottom
line from tl to bl, vertical down, unique id left
line from tr to br, vertical down, unique id right

attachment areas
point tl, label element

A.3.4 Shape *Checkbox*

primitives
line from tl to tr, horizontal right, unique id top
line from bl to br, horizontal right, unique id bottom
line from tl to bl, vertical down, unique id left
line from tr to br, vertical down, unique id right
text at polyline text_line, unique id label

computation
point where x is 2 * tr.x - tl.x, y is tr.x, unique id c_tr
point where x is 2 * br.x - bl.x, y is br.x, unique id c_br
line from c_tr to c_br, unique id text_line

constraints
distance tl-bl less than 100, hard constraint
distance tl-br less than 100, hard constraint

attachment areas
point tl, label element

A.3.5 Shape *Radiobutton*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right
text at point right, unique id label

constraints
distance left-right less than 100, hard constraint

attachment areas
point left, label element

A.3.6 Shape *Combobox*

primitives
line from tl to tm, horizontal right, unique id top_left
line from tm to tr, horizontal right, unique id top_right
line from bl to bm, horizontal right, unique id bottom_left
line from bm to br, horizontal right, unique id bottom_right
line from tl to bl, vertical down, unique id left
line from ml to ml, vertical down, unique id middle
line from tr to br, vertical down, unique id right
text within polygon top_left-middle-bottom_left-left, unique id caption

attachment areas
point tl, label element

A.3.7 Shape *HorizontalSlider*

primitives
line from tl to tr, horizontal right, unique id top
line from bl to br, horizontal right, unique id bottom
line from tl to ml, vertical down, unique id top_left
line from ml to bl, vertical down, unique id bottom_left
line from tr to mr, vertical down, unique id top_right
line from mr to br, vertical down, unique id bottom_right
line from ml to l, horizontal left, unique id left_track
line from mr to r, horizontal right, unique id right_track

text at point l, optional, unique id labelMin
text at point r, optional, unique id labelMax

constraints
distance tl-tr less than distance tl-bl, hard constraint

attachment areas
point tl, label element

A.3.8 Shape *VerticalSlider*

primitives
line from tl to tm, horizontal right, unique id top_left
line from tm to tr, horizontal right, unique id top_right
line from bl to bm, horizontal right, unique id bottom_left
line from bm to br, horizontal right, unique id bottom_right
line from tl to bl, vertical down, unique id left
line from tr to br, vertical down, unique id right
line from tm to t, vertical up, unique id top_track
line from bm to b, vertical down, unique id bottom_track
text at point b, optional, unique id labelMin
text at point t, optional, unique id labelMax

constraints
distance tl-bl less than distance tl-tr, hard constraint

attachment areas
point tl, label element

A.3.9 Postprocessing

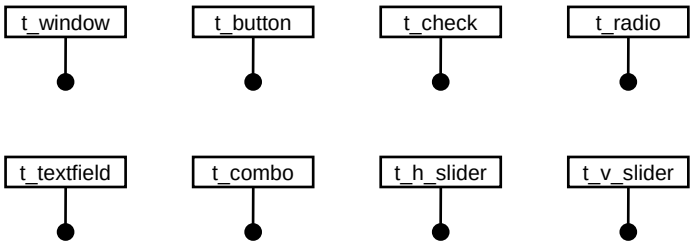
identify conflicts := true
remove larger shapes := false

A.3.10 Relation Types

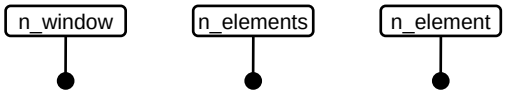
from container to element, rigid, label contains

A.3.11 Parser Symbols

terminal symbols



nonterminal symbols (start symbol is n_window)



attributes of the terminal symbols

t_window.shape for the original shape that has been recognized
t_window.caption for the caption of the window
t_button.shape for the original shape that has been recognized
t_button.caption for the caption of the button
t_check.shape for the original shape that has been recognized
t_check.label for the label of the checkbox
t_radio.shape for the original shape that has been recognized
t_radio.label for the label of the radio button
t_textfield.shape for the original shape that has been recognized
t_combo.shape for the original shape that has been recognized
t_combo.caption for the caption of the combo box
t_h_slider.shape for the original shape that has been recognized
t_h_slider.min for the minimum label of the slider
t_h_slider.max for the maximum label of the slider
t_v_slider.shape for the original shape that has been recognized
t_v_slider.min for the minimum label of the slider
t_v_slider.max for the maximum label of the slider

attributes of the nonterminal symbols

n_window.window for the resulting window, semantic attribute
n_elements.elements for the set of all elements
n_element.element for the single control element represented by this edge

A.3.12 Reduction Rules

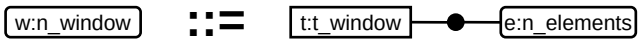
w.shape := s.self()
w.caption := s.caption

t.shape := s.self()
t.caption := s.caption

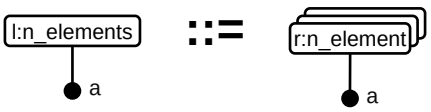
t.shape := s.self()
t.label := s.label

The next five rules are very similar to the two before, and hence omitted.
One after another they describe the reduction process for model edges representing radio buttons, text fields, combo boxes, and the two sliders.

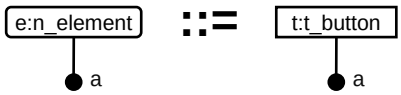
A.3.13 Production Rules



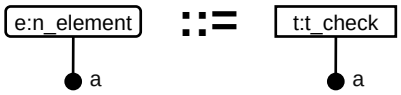
w.window := createWindow(t.shape, t.caption, e.elements)



l.elements := takeAll(r.element)



e.element := createButton(t.shape, t.caption)

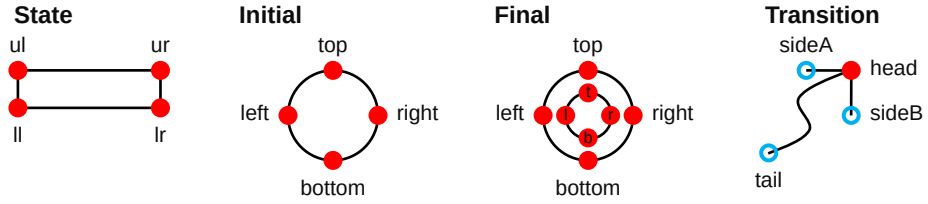


e.element := createCheckbox(t.shape, t.label)

The next five rules are very similar to the two before, and hence omitted. One after another they describe the transformation of terminal edges representing radio buttons, text fields, combo boxes, and the two sliders into nonterminal edges labeled n_element.

A.4 Statecharts

This section shows the specification for statecharts (cf. Section 2.4). The final result of sketch processing is a model of the sketched statechart, if such a model can be found.



A.4.1 Shape *State*

primitives
line from tl to tr, horizontal right, unique id line1
line from bl to tl, vertical up, unique id line2
line from br to bl, horizontal left, unique id line3
line from tr to br, vertical down, unique id line4
text within polygon line1-line2-line3-line4, unique id name

attachment areas
polyline line1-line2-line3-line4, label border
polygon line1-line2-line3-line4, label area

A.4.2 Shape *Initial*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right

attachment areas
polyline top_left-top_right-bottom_right-bottom_left, label border

A.4.3 Shape *Final*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right

arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right
arc from t to l, quadrant 2, counter-clockwise, unique id t_l
arc from t to r, quadrant 1, clockwise, unique id t_r
arc from b to l, quadrant 3, clockwise, unique id b_l
arc from b to r, quadrant 4, counter-clockwise, unique id b_r

constraints
top.y less than t.y
left.x less than l.x
right.x greater than r.x
bottom.y greater than b.y

attachment areas
polyline top_left-top_right-bottom_right-bottom_left, label border

A.4.4 Shape *Transition*

primitives
line from head to sideA, arbitrary direction, unique id lineA
line from head to sideB, arbitrary direction, unique id lineB
link from head to tail, unique id shaft
text at polyline shaft, optional, unique id label

constraints
distance head-tail greater than 80, soft constraint
distance head-sideA greater than 50, soft constraint
distance head-sideB greater than 50, soft constraint
distance head-sideA less than distance head-tail, hard constraint
distance head-sideB less than distance head-tail, hard constraint
angle sideA-head-tail greater than 20°, hard constraint
angle sideA-head-tail less than 70°, hard constraint
angle tail-head-sideB greater than 20°, hard constraint
angle tail-head-sideB less than 70°, hard constraint

attachment areas
point head, label trans
point tail, label trans

A.4.5 Postprocessing

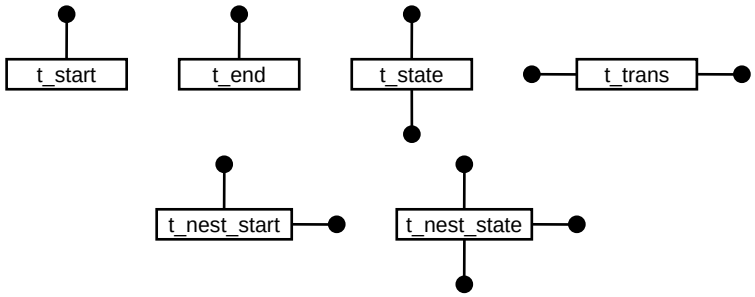
identify conflicts := true
remove larger shapes := false

A.4.6 Relation Types

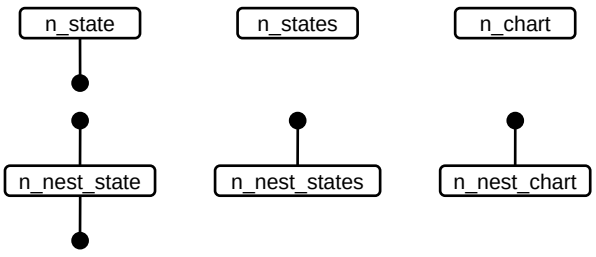
from area(s1) to border(s2), rigid, label contains,
condition s1.width > s2.width where
s1.width is s1.tr.x - s1.tl.x,
s2.width is s2.tr.x - s2.tl.x if s2 is a state,
s2.width is s2.right.x - s2.left.x if s2 is an initial
from trans to border, not rigid, label at

A.4.7 Parser Symbols

terminal symbols



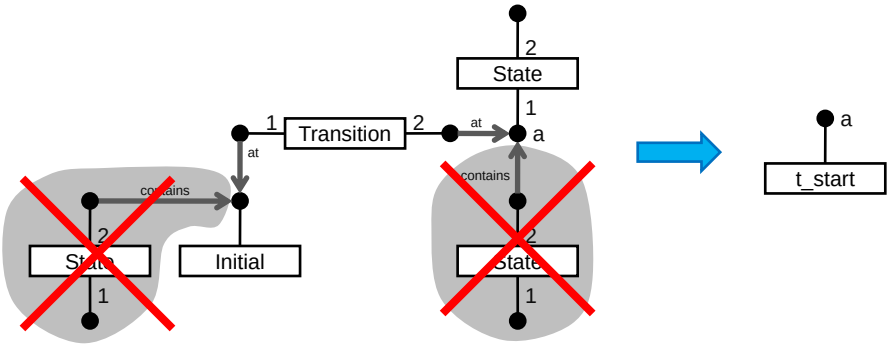
nonterminal symbols (start symbol is n_chart)

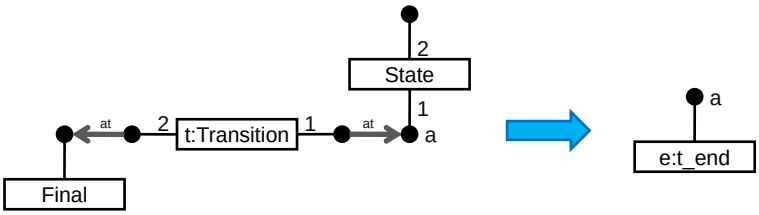


attributes of the terminal symbols
t_shape.name for the name of the shape
t_trans.label for the label of the transition
t_end.label for the label of the transition to a final state

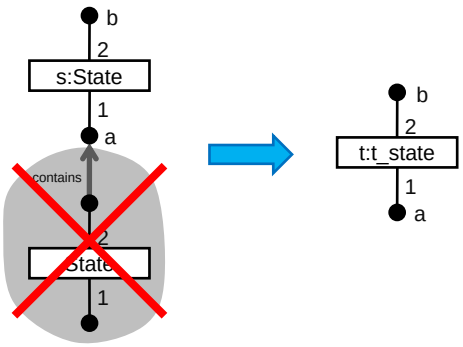
attributes of the nonterminal symbols
n_state.state for the represented state
n_nest_state.state for the represented state
n_states.states for the represented states
n_nest_states.states for the represented states
n_chart.chart for the represented state chart, semantic attribute
n_nest_chart.chart for the represented state chart

A.4.8 Reduction Rules

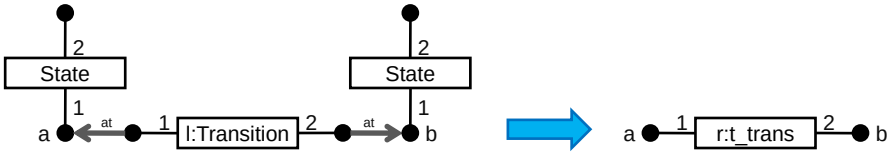




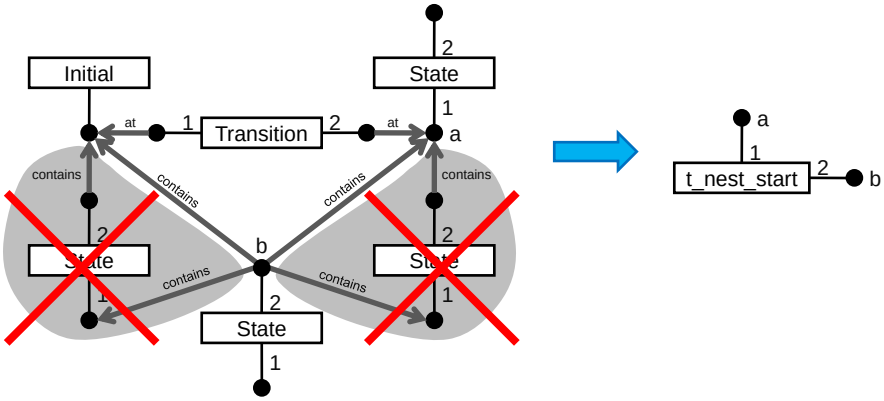
e.label := t.label

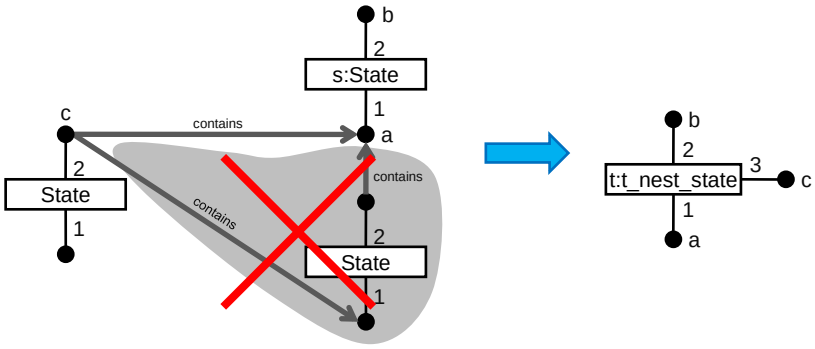


t.name := s.name



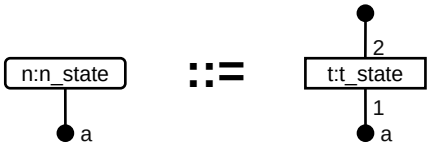
l.label := r.label



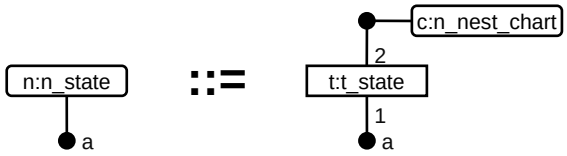


t.name := s.name

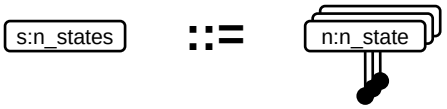
A.4.9 Production Rules



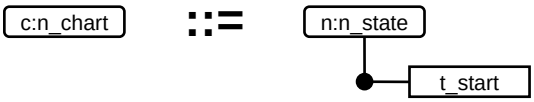
n.state := createState(t.name)



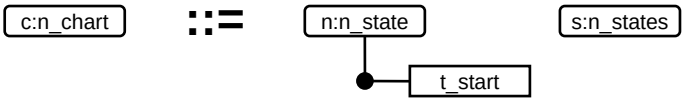
n.state := createState(t.name, c.chart)



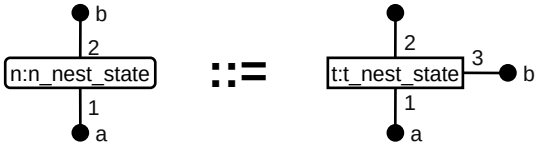
s.states := takeAll(n.state)



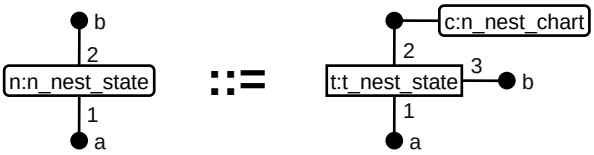
c.chart := createChart(n.state)



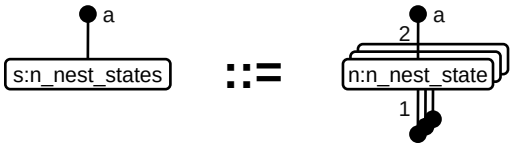
c.chart := createChart(n.state, s.states)



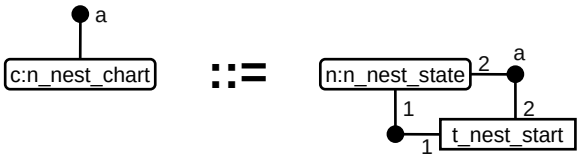
n.state := createState(t.name)



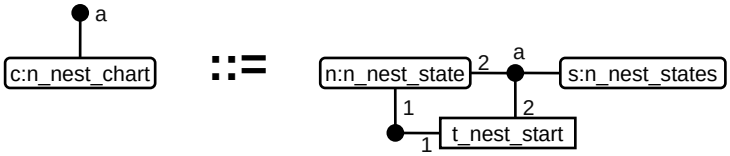
n.state := createState(t.name, c.chart)



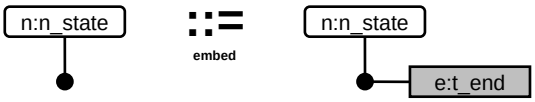
s.states := takeAll(n.state)



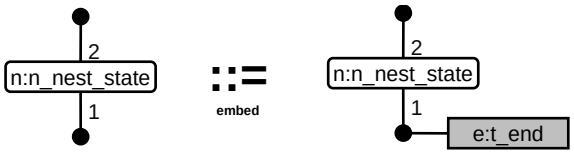
c.chart := createChart(n.state)



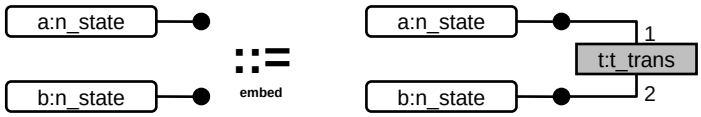
c.chart := createChart(n.state, s.states)



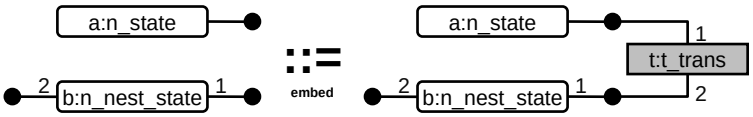
setFinalTransition(n.state, e.label)



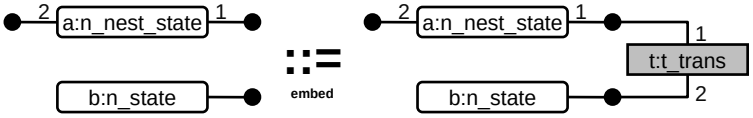
setFinalTransition(n.state, e.label)



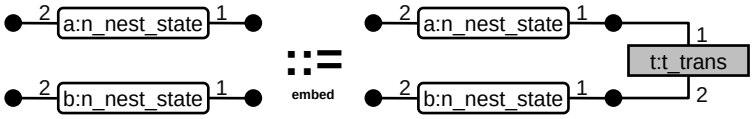
addTransition(a.state, b.state, t.label)



addTransition(a.state, b.state, t.label)



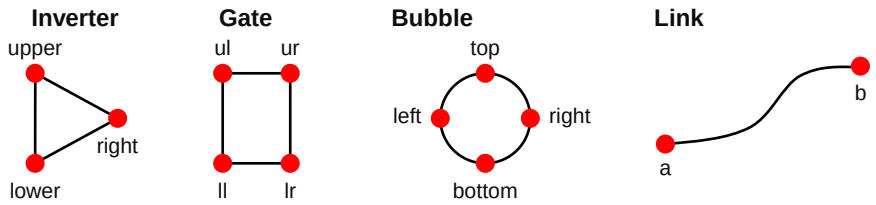
addTransition(a.state, b.state, t.label)



addTransition(a.state, b.state, t.label)

A.5 Boolean Logic Diagrams

This section shows the specification for BLDs, which are illustrated in Section 2.5. The final result of sketch processing is a boolean logic expression.



A.5.1 Shape *Inverter*

primitives
line from upper to right, arbitrary direction, unique id top
line from upper to lower, vertical down, unique id left
line from lower to right, arbitrary direction, unique id bottom

constraints
angle lower-upper-right greater than 45°, hard constraint
angle lower-upper-right less than 90°, hard constraint
angle right-lower-upper greater than 45°, hard constraint
angle right-lower-upper less than 90°, hard constraint

attachment areas
polyline left, label input
point right, label output

A.5.2 Shape *Gate*

primitives
line from ul to ur, horizontal right, unique id top
line from ll to ul, vertical up, unique id left
line from lr to ll, horizontal left, unique id bottom
line from ur to lr, vertical down, unique id right
text within polygon top-left-bottom-right, regex `&|1|>=1`, unique id label

attachment areas
polyline left, label input
polyline right, label output

A.5.3 Shape *Bubble*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right

attachment areas
point left, label bubbleInput
point right, label output

A.5.4 Shape *Link*

primitives
link from a to b, unique id shaft
text at point a, optional, regex `[a-zA-Z].*`, unique id nameA
text at point b, optional, regex `[a-zA-Z].*`, unique id nameB

attachment areas
point a, label linkEnd
point b, label linkEnd

A.5.5 Postprocessing

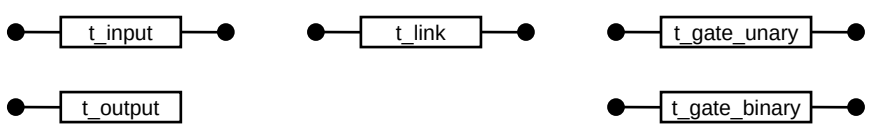
identify conflicts := true
remove larger shapes := false

A.5.6 Relation Types

from linkEnd to input, not rigid, label inputLink
from output to linkEnd, not rigid, label outputLink
from output to bubbleInput, not rigid, label hasBubble

A.5.7 Parser Symbols

terminal symbols



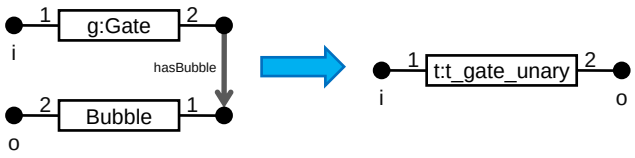
nonterminal symbols (start symbol is n_start)

attributes of the terminal symbols
t_input.var for the name of the input variable
t_input.y for the y-value of the point where the input is connected
t_output.var for the name of the output variable
t_link.y for the y-value of the point where the link is connected as input
t_gate_unary.op for the operator that is represented
t_gate_unary.invert to describe whether the gate is inverted
t_gate_binary.op for the operator that is represented
t_gate_binary.invert to describe whether the gate is inverted

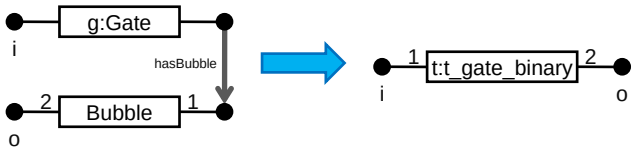
attributes of the nonterminal symbols
n_gate.expr for the represented expression
n_start.expr for the represented expression, semantic attribute

A.5.8 Reduction Rules

Condition g.label = ”&” OR g.label = ”>=1”
t.op := g.label
t.invert := false

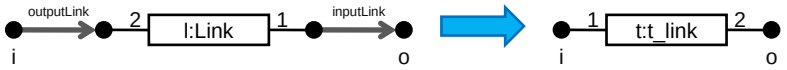


Condition g.label = ”1”
t.op := g.label
t.invert := true

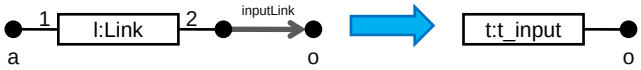


Condition g.label = ”&” OR g.label = ”>=1”
t.op := g.label
t.invert := true

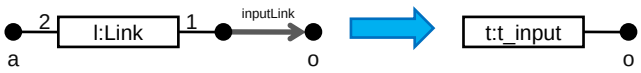
t.y := l.self().b.y



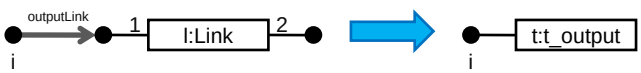
t.y := l.self().a.y



Condition l.nameA is set
t.var := l.nameA
t.y := l.self().b.y

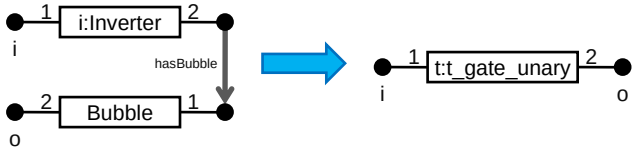


Condition l.nameB is set
t.var := l.nameB
t.y := l.self().a.y



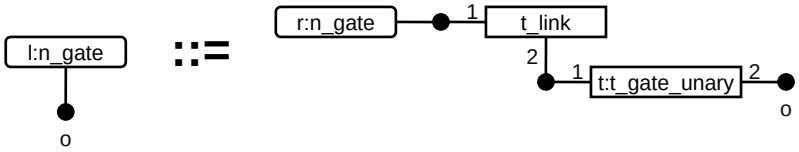
Condition l.nameB is set
t.var := l.nameB

Condition l.nameA is set
t.var := l.nameA

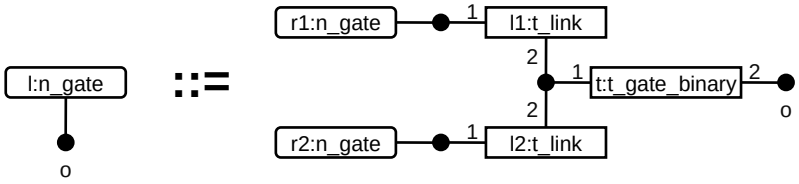


t.op := g.label
t.invert := true

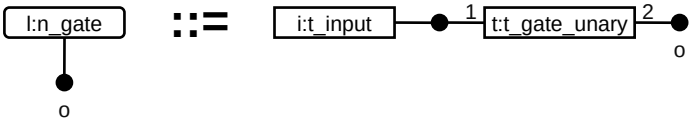
A.5.9 Production Rules



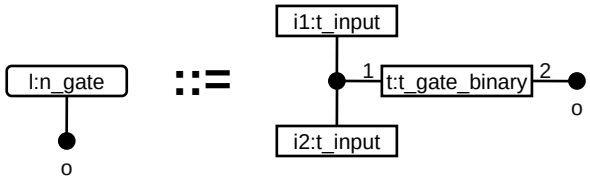
l.expr := createUnaryExpr(t.op, t.invert, r.expr)



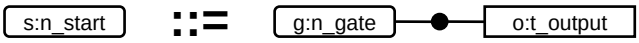
Condition l1.y $<$ l2.y
l.expr := createBinaryExpr(t.op, t.invert, r1.expr, r2.expr)



l.expr := createUnaryExprWithInput(t.op, t.invert, i.var)



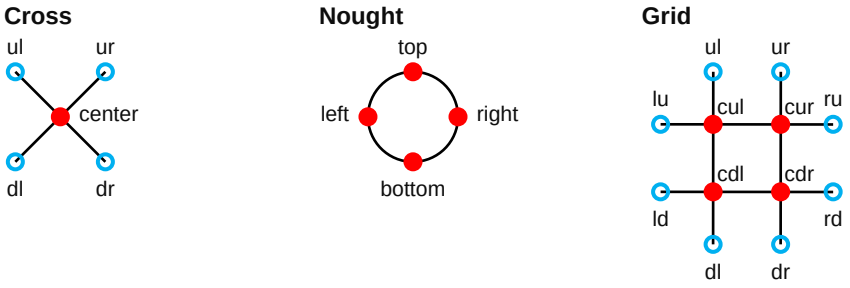
Condition i1.y $<$ i2.y
l.expr := createBinaryExprWithInput(t.op, t.invert, i1.var, i2.var)



s.expr := o.var + ” := ” + g.expr

A.6 Tic-tac-toe

This section shows the specification for Tic-tac-toe (cf. Section 2.6). The final result of sketch processing is a statement about the sketched game situation.



A.6.1 Shape *Cross*

primitives
line from center to ul, diagonal up left, unique id ul
line from center to ur, diagonal up right, unique id ur
line from center to dl, diagonal down left, unique id dl
line from center to dr, diagonal down right, unique id dr

attachment areas
point center, label mark

A.6.2 Shape *Nought*

primitives
arc from top to left, quadrant 2, counter-clockwise, unique id top_left
arc from top to right, quadrant 1, clockwise, unique id top_right
arc from bottom to left, quadrant 3, clockwise, unique id bottom_left
arc from bottom to right, quadrant 4, counter-clockwise, unique id bottom_right

computation
point where x is (top.x + bottom.x) / 2, y is (top.y + bottom.y) / 2, unique id center

attachment areas
point center, label mark

A.6.3 Shape *Grid*

primitives
line from cul to cur, vertical right, unique id lcu
line from cur to cdr, horizontal down, unique id lcr
line from cdr to cdl, vertical left, unique id lcd
line from cdl to cul, horizontal up, unique id lcl
line from cul to ul, horizontal up, unique id lul
line from cur to ur, horizontal up, unique id lur
line from cdl to dl, horizontal down, unique id ldl
line from cdr to dr, horizontal down, unique id ldr
line from cul to lu, vertical left, unique id llu
line from cdl to ld, vertical left, unique id lld
line from cur to ru, vertical right, unique id lru
line from cdr to rd, vertical right, unique id lrd

computation
point where x is lu.x, y is ul.x, unique id c_ul
point where x is ru.x, y is ur.x, unique id c_ur
point where x is ld.x, y is dl.x, unique id c_dl
point where x is rd.x, y is dr.x, unique id c_dr
line from ur to c_ur, unique id c_uru
line from dr to c_dr, unique id c_drd
line from dl to c_dl, unique id c_dld
line from ul to c_ul, unique id c_ulu

attachment areas
polygon lul-lcu-lur, label board
polygon c_uru-lur-lru, label board
polygon lru-lcr-lrd, label board
polygon lrd-ldr-c_drd, label board
polygon ldr-lcd-ldl, label board
polygon lld-ldl-c_dld, label board
polygon llu-lcl-lld, label board
polygon c_ulu-lul-llu, label board
polygon lcu-lcr-lcd-lcl, label board

A.6.4 Postprocessing

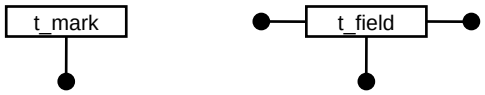
identify conflicts := true
remove larger shapes := false

A.6.5 Relation Types

from mark to board, rigid, label at

A.6.6 Parser Symbols

terminal symbols



nonterminal symbols (start symbol is n_grid)

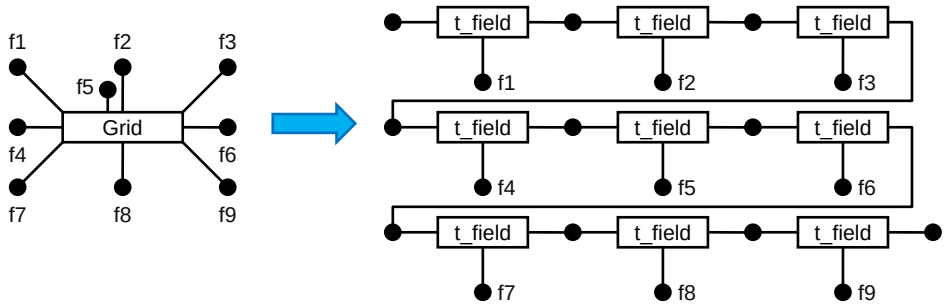
attributes of the terminal symbols
t_mark.type for the type of mark (either cross or nought)

attributes of the nonterminal symbols
n_field.mark for the mark contained in the field
n_grid.config for complete configuration of all nine marks
n_grid.result for the statement about the sketched game configuration, semantic attribute

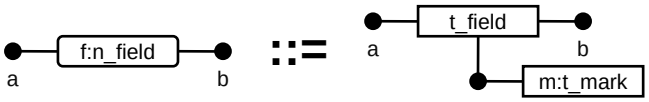
A.6.7 Reduction Rules

m.type := cross

m.type := nought

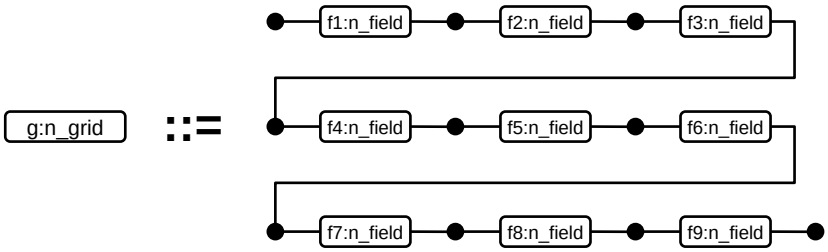


A.6.8 Production Rules



f.mark := m.type

f.mark := createEmptyMark()

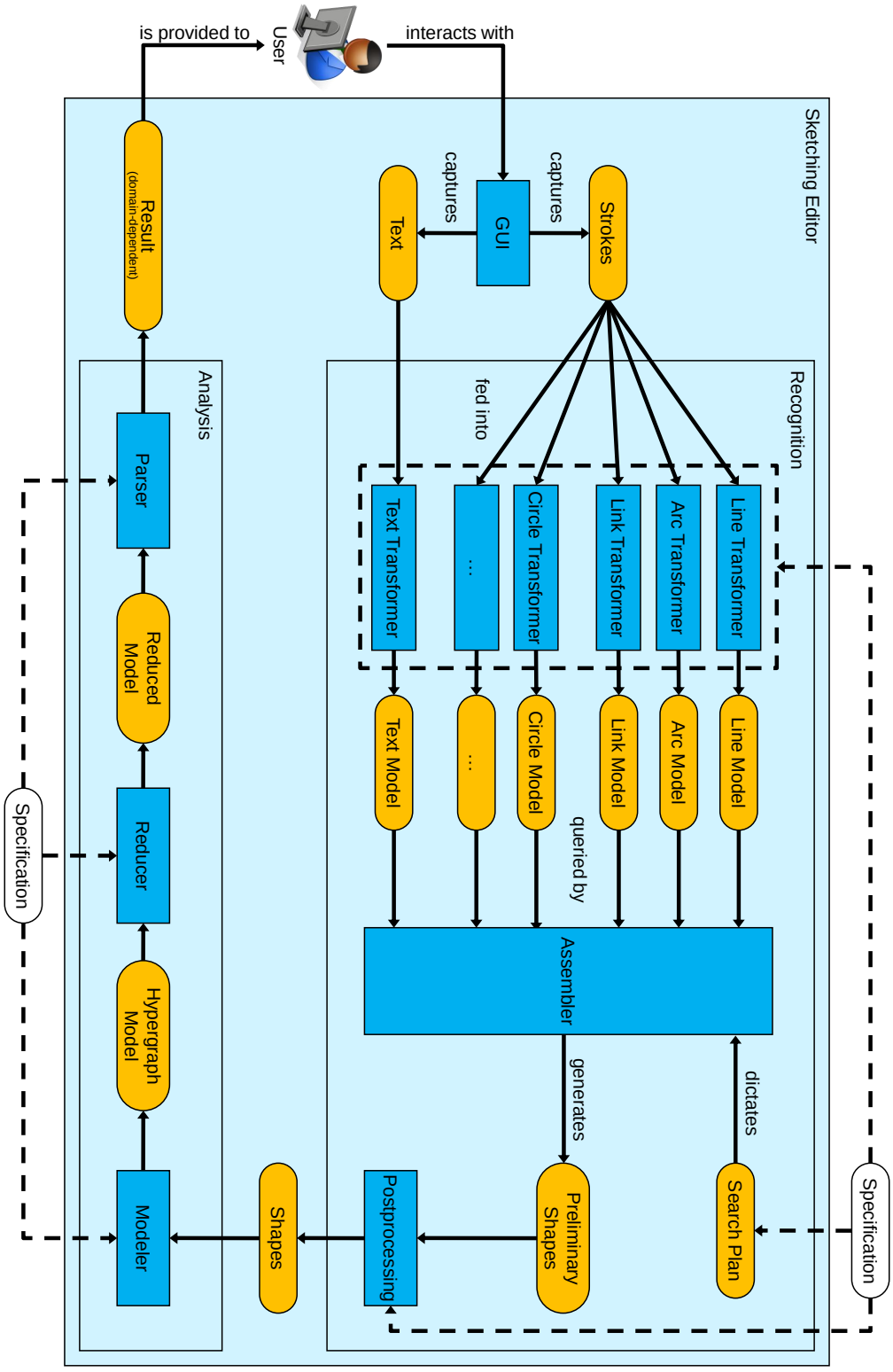


g.config := createCompleteConfig(f1.mark, f2.mark, f3.mark, f4.mark, f5.mark, f6.mark, f7.mark, f8.mark, f9.mark)
g.result := checkConfig(g.config)

Appendix B

The Complete Concept

The figure on the next page shows all processing units and data structures comprising the complete processing chain. This means that it merges all figures that show only partially the processing chain. Note that the specification is shown two times for the sake of a simpler layout only.


User
is provided to
interacts with
Sketching Editor
Result
(domain-dependent)
Text
captures
GUI
captures
Strokes
Recognition
fed into
Text Transformer
...
Circle Transformer
Link Transformer
Arc Transformer
Line Transformer
Text Model
...
Circle Model
Link Model
Arc Model
Line Model
queried by
Assembler
generates
dictates
Preliminary Shapes
Search Plan
Specification
Postprocessing
Shapes
Analysis
Parser
Reduced Model
Reducer
Hypergraph Model
Modeler
Specification

Appendix C

Detailed Example

This chapter gives an impression of how a real sketch is processed by the implementation. Figure C.1 shows a simple Petri net which is used as running example. The following two sections describe the processing results obtained by the recognition stage and the analysis stage. Note that all figures in this chapter are either screenshots from the implementation, or exact reproductions.

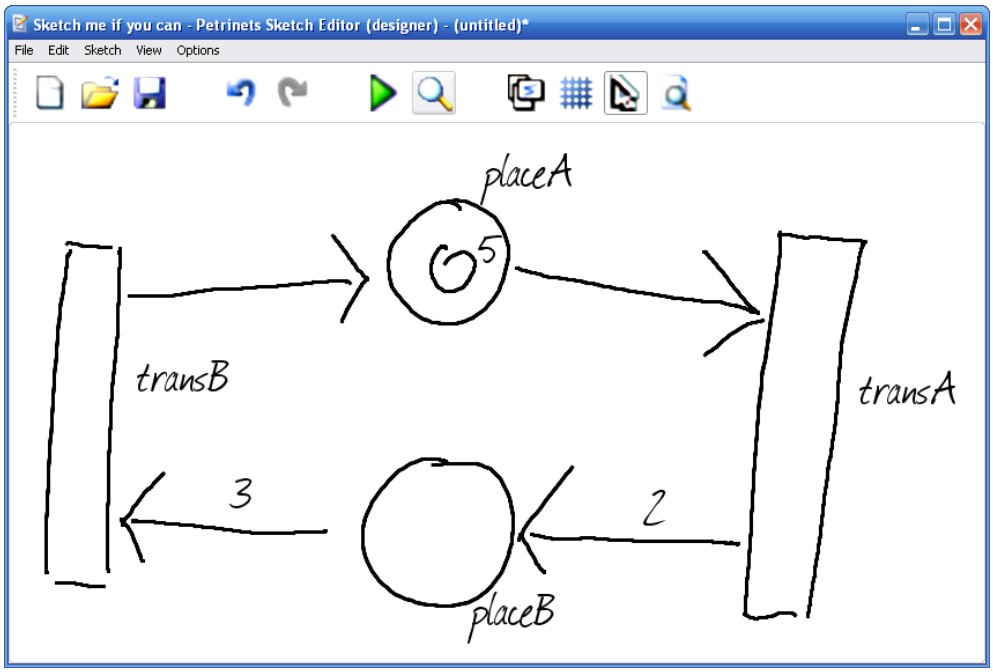
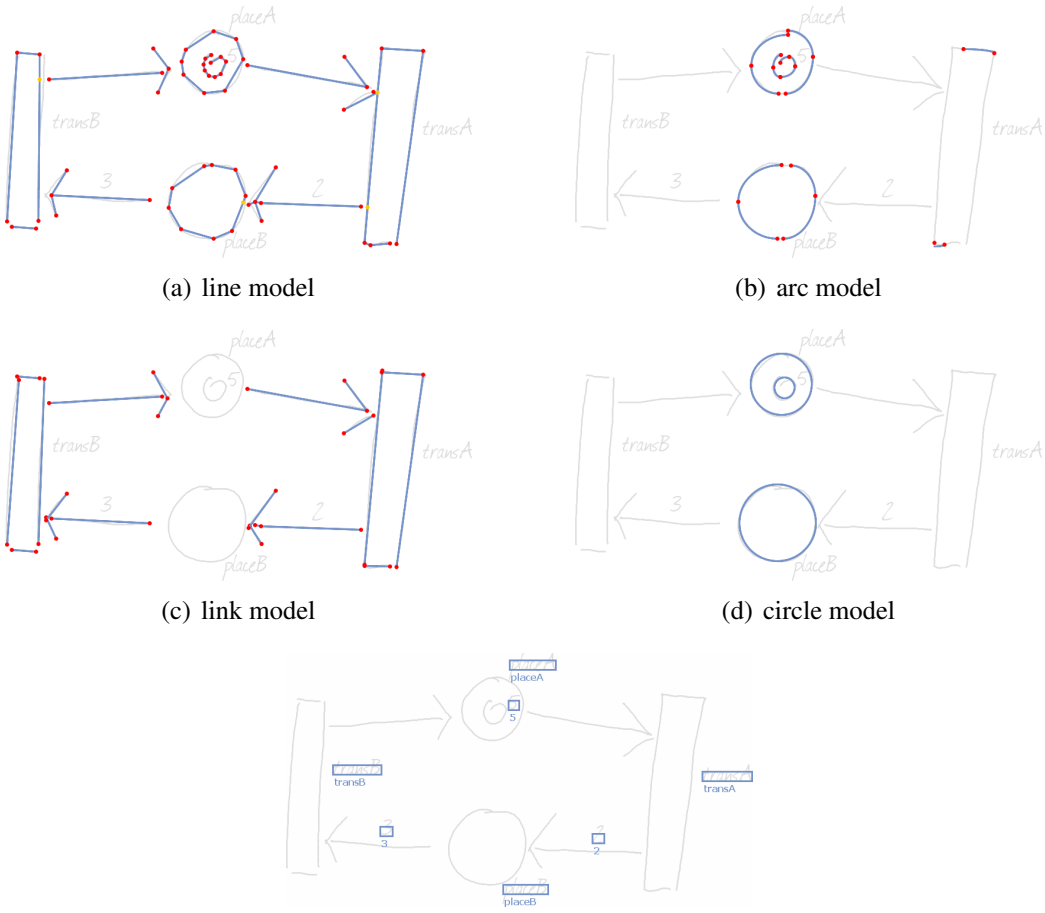


Figure C.1: A simple Petri net used as running example.

C.1 Recognition Stage

The first step in the recognition stage is preprocessing by the transformers. They yield the five different models illustrated in Chapter 4. Figure C.2 shows the five models.



(a) line model

(b) arc model

(c) link model

(d) circle model

(e) text model

Figure C.2: Contents of the models generated for the sketch shown in Figure C.1. The sketch itself is grayed out.

The assembler recognizes 26 shapes from the data contained in the models: two transitions, nine places, nine tokens, and six arrows. Figure C.3 shows these 26 shapes. However, as each circle is recognized as both a place and a token, the figure only shows places in order to be less cluttered up. Note that tokens, unlike places, do not consist of any text.

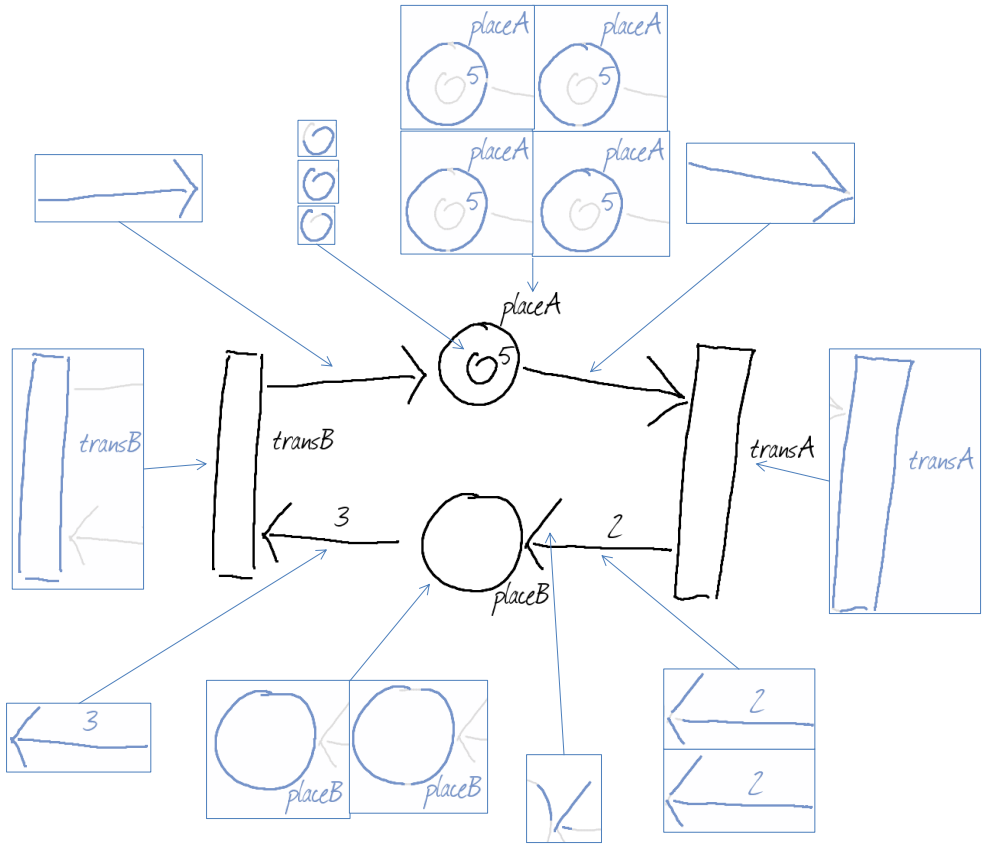


Figure C.3: 17 of the 26 shapes recognized from the models in Figure C.2. The nine tokens corresponding to the nine places are omitted.

The recognition stage's final step is postprocessing. It has two tasks in the case of Petri nets: removal of duplicates and identification of conflicts. The result of this step is shown in Figure C.4. As before, dashed lines indicate conflicts. It can be seen that the removal of duplicates cuts the number of shapes in half, resulting in 13 shapes that are passed on to the analysis step. Compared to the nine shapes that make the original sketch, 13 is a good figure. The four extra shapes are one falsely recognized arrow, two tokens similar to places, and one place similar to a token.

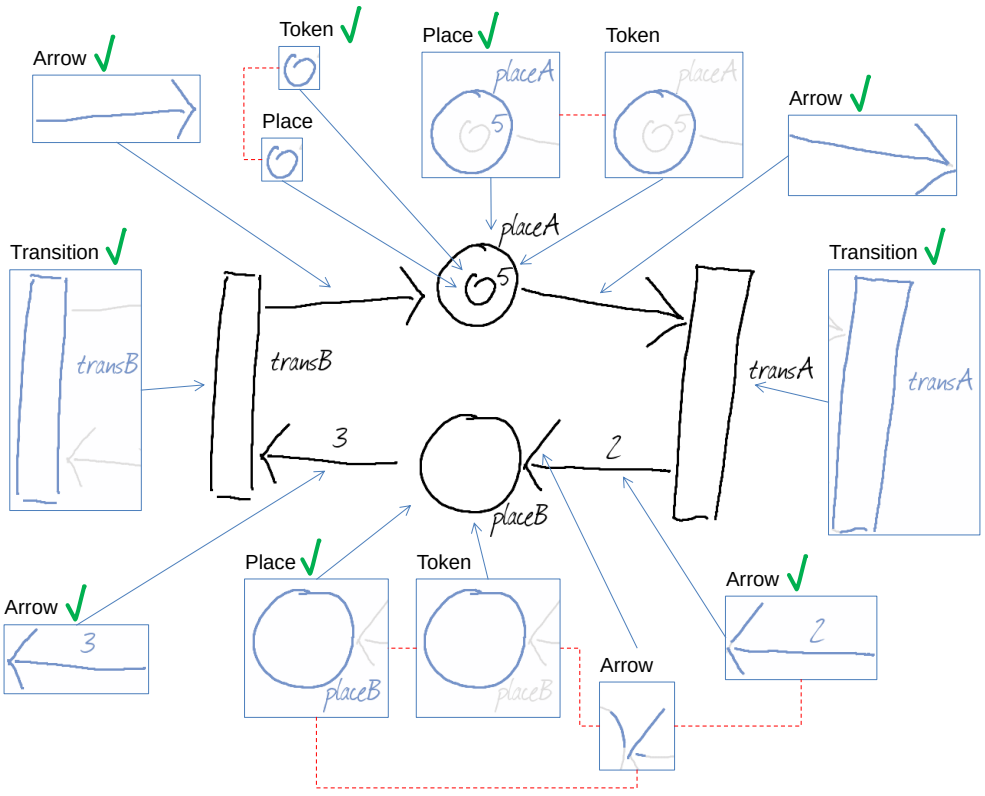


Figure C.4: The 13 shapes left after removal of duplicates, and the conflicts between them. The check marks indicate which of the shapes are correct. Shapes without check marks are false positives.

C.2 Analysis Stage

In the first step of the analysis stage, the modeler creates the HM shown in Figure C.5. The figure additionally shows the attributes of all shape edges. Obviously, the figure's structure is very similar to that of Figure C.4.

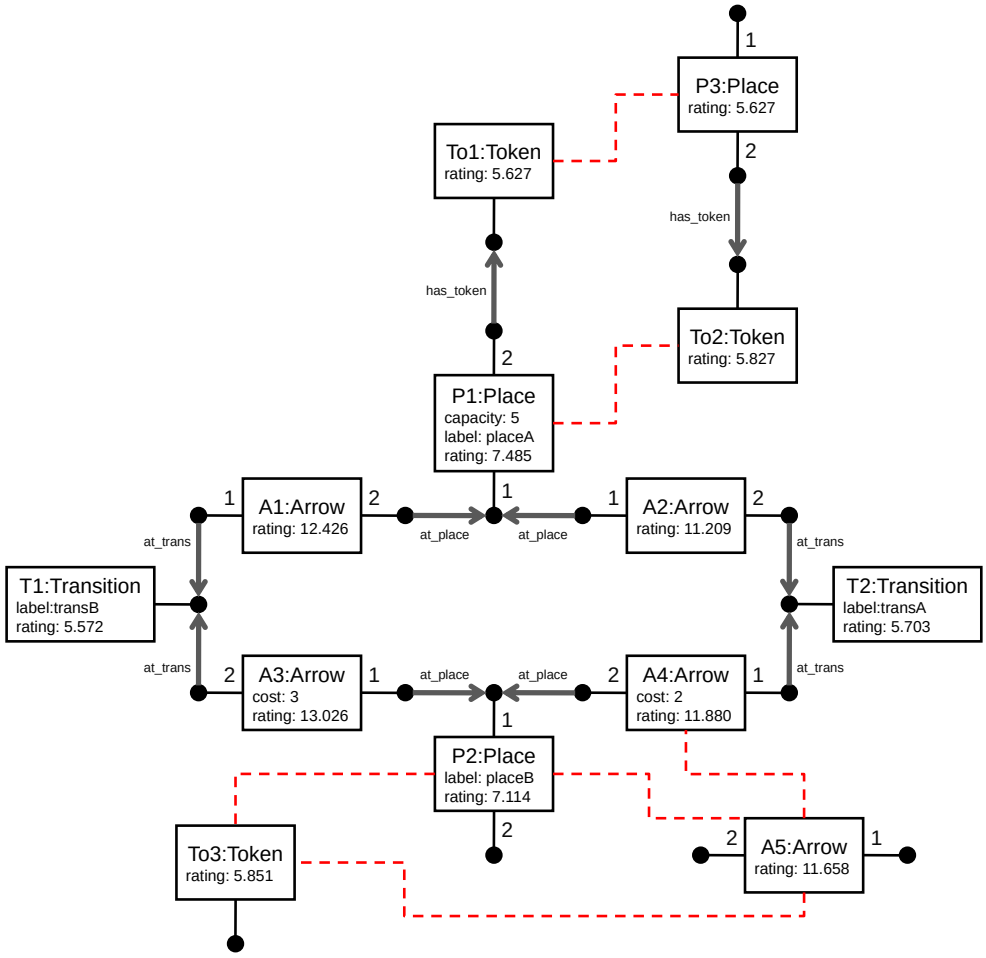


Figure C.5: The HM for the sketch shown in Figure C.1.

The reducer then processes the HM and produces the RHM shown in Figure C.6. In doing so, the number of edges is reduced from 23 to ten, and the number of nodes is reduced from 21 to five. Instead of six conflicts in the HM, there is only one in the RHM. The reason is that some of the conflicting edges are not reduced. The token To2 is not reduced because it is not contained in a (larger) place, and so is To3. Arrow A5 is not reduced because it does not connect a place and a transition.

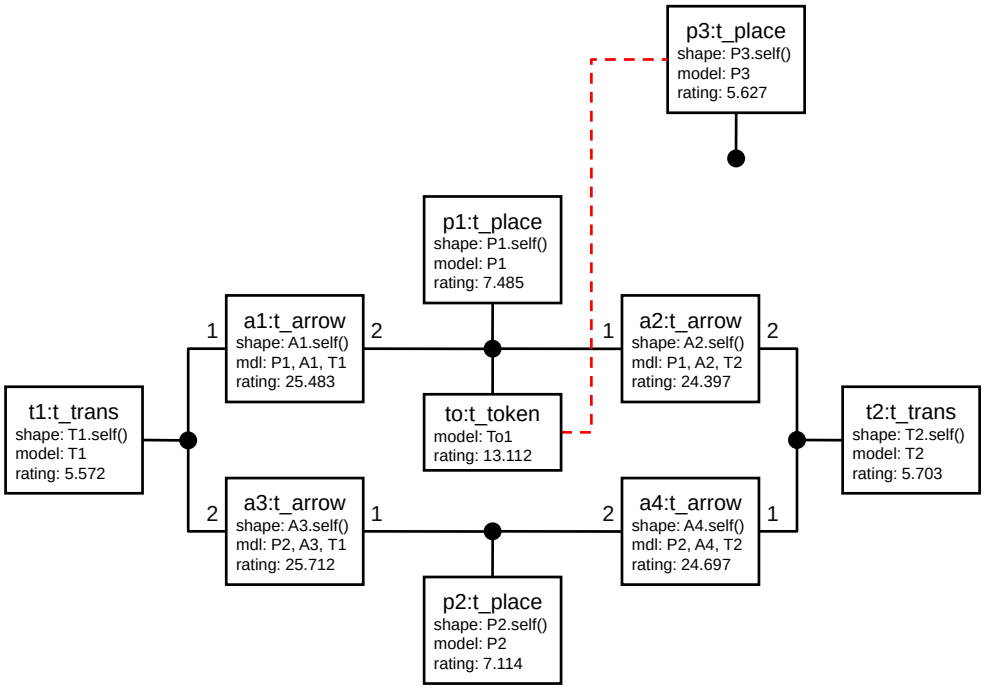


Figure C.6: The RHM for the HM shown in Figure C.5. The attribute model is sometimes abbreviated as mdl.

Figure C.7 shows the derivation DAG that is computed by the parser. The conflict between p3 and to is carried over to N1 and N2. After the start symbol N8 is reached, N1 is removed from the set production, as its weighted rating is considerably less than the weighted rating of N2, as the latter has a greater rating and one of its children is part of the context of two embedded edges (a1 and a2). Accordingly, the ratings of N6 and N8 do not include the rating of N1. For N8 the values in parentheses include the four embedded arrows.

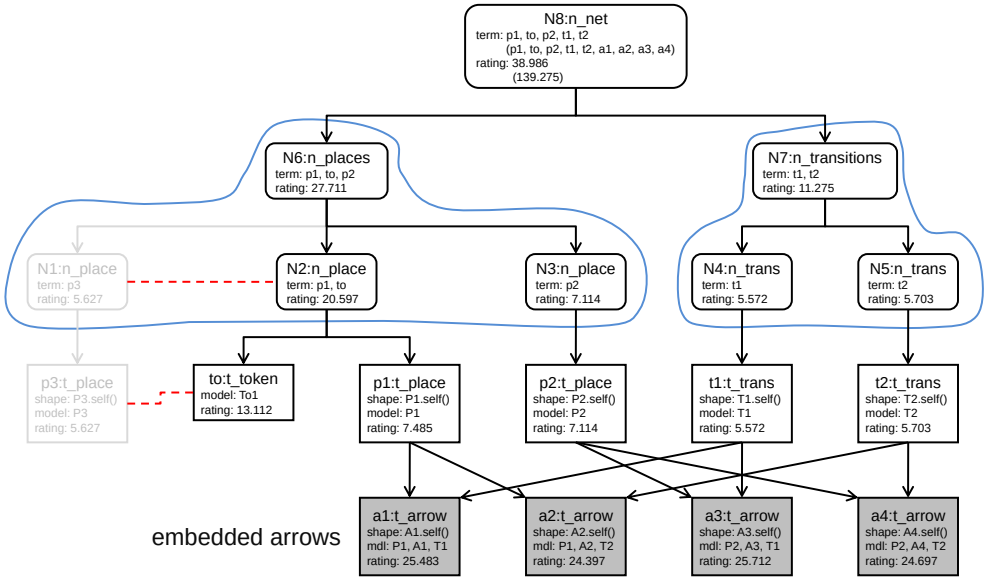


Figure C.7: The DAG created by the parser for the RHM shown in Figure C.6. Set productions are encircled, edges removed from sets are grayed out.

Based on the DAG shown in Figure C.7 the semantic attribute of the start symbol N8 is evaluated. This can be accomplished by the following evaluation order:

N2.model := createPlaceWithToken(p1.shape)
N3.model := createPlace(p2.shape)
N4.model := createTransition(t1.shape)
N5.model := createTransition(t2.shape)
N6.set := {N2, N3}
N7.set := {N4, N5}
N8.net := createNet(N8.set, N7.set)

addArrow(t1.shape, a1.shape, p1.shape)
addArrow(p1.shape, a2.shape, t2.shape)
addArrow(p2.shape, a3.shape, t1.shape)
addArrow(t2.shape, a4.shape, p2.shape)

Depending on how the illustrated functions calls work, a model of the Petri net could be obtained. A textual representation may look like this:

Petri net, 2 places, 2 transitions
==========
Place ”placeA”, capacity 5, contains a token
Place ”placeB”
Transition ”transA”
Transition ”transB”
Arrow from ”transB” to ”placeA”
Arrow from ”placeA” to ”transA”
Arrow from ”transA” to ”placeB”, cost 2
Arrow from ”placeB” to ”transB”, cost 3
==========

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*. Prentice Hall, 1972.

[2] C. Alvarado. *Multi-Domain Sketch Understanding*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[3] C. Alvarado and R. Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI '01)*, pages 1365–1374, August 2001.

[4] C. Alvarado and R. Davis. SketchREAD: a multi-domain sketch recognition engine. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*, pages 23–32, New York, NY, USA, 2004. ACM.

[5] C. Alvarado and R. Davis. Dynamically constructed bayes nets for multi-domain sketch understanding. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI '05)*, pages 1407–1412. Professional Book Center, 2005.

[6] A. Apte, V. Vo, and T. D. Kimura. Recognizing multistroke geometric shapes: an experimental evaluation. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST '93)*, pages 121–128, New York, NY, USA, 1993. ACM.

[7] J. Arvo and K. Novins. Appearance-preserving manipulation of hand-drawn graphs. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '05)*, pages 61–68, New York, NY, USA, 2005. ACM.

[8] S.-H. Bae, R. Balakrishnan, and K. Singh. ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proceedings of*

the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08), pages 151–160, New York, NY, USA, 2008. ACM.

[9] C. Berge. *Graphs and Hypergraphs*. North Holland, Amsterdam, the Netherlands, 1973.

[10] C. M. Bishop, M. Svensen, and G. E. Hinton. Distinguishing text from graphics in on-line handwritten ink. In *Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition (IWFHR '04)*, pages 142–147, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. Blostein. General diagram-recognition methodologies. In *Selected Papers from the 1st International Workshop on Graphics Recognition. Methods and Applications*, pages 106–122, London, UK, 1996. Springer-Verlag.

[12] D. Blostein, E. Lank, A. Rose, and R. Zanibbi. User interfaces for online diagram recognition. In *Selected Papers from the 4th International Workshop on Graphics Recognition Algorithms and Applications (GREC '01)*, pages 92–103, London, UK, 2002. Springer-Verlag.

[13] C. Bongartz. Übersicht über Sketching: aktuelle Ansätze und Systeme im Vergleich. Universität der Bundeswehr München, 2008. Diploma thesis, UniBwM-ID 14/2007, only available in German.

[14] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge. JavaSketchIt: Issues in sketching the look of user interfaces. In *Papers from the 2002 AAAI Spring Symposium on Sketch Understanding*, pages 9–14, Menlo Park, CA, USA, 2002. AAAI Press.

[15] C. Calhoun, T. F. Stahovich, T. Kurtoglu, and L. B. Kara. Recognizing multi-stroke symbols. In *Papers from the 2002 AAAI Spring Symposium on Sketch Understanding*, pages 15–23. AAAI Press, Menlo Park, USA, 2002.

[16] G. Casella, G. Costagliola, V. Deufemia, M. Martelli, and V. Mascardi. An agent-based framework for context-driven interpretation of symbols in diagrammatic sketches. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '06)*, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[17] G. Casella, V. Deufemia, and V. Mascardi. A multi-agent system for hand-drawn diagram recognition. *Proceedings of the 9th International Conference on Document Analysis and Recognition (ICDAR '07)*, 2:739–743, September 2007.

[18] G. Casella, V. Deufemia, V. Mascardi, G. Costagliola, and M. Martelli. An agent-based framework for sketched symbol interpretation. *Journal of Visual Languages & Computing*, 19(2):225–257, 2008.

[19] R. Chung, P. Mirica, and B. Plimmer. InkKit: a generic design tool for the tablet PC. In *Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction (CHINZ '05)*, pages 29–30, New York, NY, USA, 2005. ACM.

[20] G. Costagliola, V. Deufemia, G. Polese, and M. Risi. A parsing technique for sketch recognition systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC '04)*, pages 19–26, Washington, DC, USA, 2004. IEEE Computer Society.

[21] G. Costagliola, V. Deufemia, and M. Risi. Sketch grammars: A formalism for describing and recognizing diagrammatic sketch languages. In *Proceedings of the 8th International Conference on Document Analysis and Recognition (ICDAR '05)*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[22] G. Costagliola, V. Deufemia, and M. Risi. A trainable system for recognizing diagrammatic sketch languages. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '05)*, pages 281–283, Washington, DC, USA, 2005. IEEE Computer Society.

[23] G. Costagliola, V. Deufemia, and M. Risi. A multi-layer parsing strategy for on-line recognition of hand-drawn diagrams. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '06)*, pages 103–110, Washington, DC, USA, 2006. IEEE Computer Society.

[24] G. Costagliola, V. Deufemia, and M. Risi. Using error recovery techniques to improve sketch recognition accuracy. In W. Liu, J. Lladós, and J.-M. Ogier, editors, *Proceedings of the 7th International Workshop on Graphics Recognition. Recent Advances and New Opportunities (GREC '07)*, volume 5046 of *Lecture Notes in Computer Science*, pages 157–168. Springer, September 2007.

[25] G. Costagliola, V. Deufemia, and M. Risi. Using grammar-based recognizers for symbol completion in diagrammatic sketches. In *Proceedings of the 9th International Conference on Document Analysis and Recognition*

(ICDAR '07), volume 2, pages 1078–1082, Washington, DC, USA, 2007. IEEE Computer Society.

[26] G. Costagliola, R. Francese, M. Risi, G. Scanniello, and A. D. Lucia. A component-based visual environment development process. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pages 327–334, New York, NY, USA, 2002. ACM.

[27] G. Costagliola, A. D. Lucia, and S. Orefice. Towards efficient parsing of diagrammatic languages. In *Proceedings of the Workshop on Advanced Visual Interfaces (AVI '94)*, pages 162–171, New York, NY, USA, 1994. ACM.

[28] G. Costagliola, A. D. Lucia, S. Orefice, and G. Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages & Computing*, 13(6):573–600, 2002.

[29] G. Costagliola and G. Polese. Extended positional grammars. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, pages 103–110, Washington, DC, USA, 2000. IEEE Computer Society.

[30] A. Coyette, S. Kieffer, and J. M. Vanderdonckt. Multi-fidelity prototyping of user interfaces. In *Proceedings of the 13th International Conference on Human-Computer Interaction (INTERACT '07)*, volume 4662 of *Lecture Notes in Computer Science*, pages 150–164. Springer, September 2007.

[31] A. Coyette and J. M. Vanderdonckt. A sketching tool for designing anyuser, anyplatform, anywhere user interfaces. In *Proceedings of the 12th International Conference on Human-Computer Interaction (INTERACT '05)*, pages 550–564. Springer Verlag, 2005.

[32] A. Coyette, J. M. Vanderdonckt, and Q. Limbourg. SketchiXML: A design tool for informal user interface rapid prototyping. In *Proceedings of the 3rd International Workshop on Rapid Integration of Software Engineering Techniques (RISE '06)*, pages 160–176. Springer, September 2006.

[33] J. Davis, M. Agrawala, E. Chuang, Z. Popović, and D. Salesin. A sketching interface for articulated figure animation. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '03)*, pages 320–328, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[34] J. de Lara and H. Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE '02)*, pages 174–188, London, UK, 2002. Springer.

[35] P. Domingos and M. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *Proceedings of the 13th International Conference on Machine Learning (ICML '96)*, pages 105–112. Morgan Kaufmann, 1996.

[36] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[37] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98)*, pages 296–309, London, UK, 2000. Springer.

[38] M. J. Fonseca, C. Pimentel, and J. A. Jorge. CALI: an online scribble recognizer for calligraphic interfaces. In *Papers from the 2002 AAAI Spring Symposium on Sketch Understanding*, pages 51–58, Menlo Park, CA, USA, 2002. AAAI Press.

[39] I. J. Freeman and B. Plimmer. Connector semantics for sketched diagram recognition. In *Proceedings of the 8th Australasian User Interface Conference (AUIC '07)*, pages 71–78, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[40] L. Gennari, L. B. Kara, T. F. Stahovich, and K. Shimada. Combining geometry and domain knowledge to interpret hand-drawn diagrams. *Computers & Graphics*, 29(4):547–562, 2005.

[41] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs, Second Edition (Annals of Discrete Mathematics, Vol. 57)*. North-Holland Publishing Co., Amsterdam, the Netherlands, 2004.

[42] M. D. Gross and E. Y.-L. Do. Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of the 9th Annual ACM symposium on User Interface Software and Technology (UIST '96)*, pages 183–192, New York, NY, USA, 1996. ACM.

[43] M. D. Gross and E. Y.-L. Do. Drawing on the back of an envelope: a framework for interacting with application programs by freehand drawing. *Computers & Graphics*, 24(6):835–849, 2000.

[44] J. Grundy and J. Hosking. Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 282–291, Washington, DC, USA, May 2007. IEEE Computer Society.

[45] J. Grundy, J. Hosking, N. Zhu, and N. Liu. Generating domain-specific visual language editors from high-level tool specifications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society.

[46] T. Hammond and R. Davis. Ladder: A language to describe drawing, display, and editing in sketch recognition. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI '03)*, pages 461–467, August 2003.

[47] T. Hammond and R. Davis. Automatically transforming symbolic shape descriptions for use in sketch recognition. In *The 19th National Conference on Artificial Intelligence (AAAI '04)*, Menlo Park, CA, USA, July 2004. AAAI Press.

[48] T. Hammond and R. Davis. LADDER, a sketching language for user interface developers. *Computers & Graphics*, 29(4):518–532, 2005.

[49] T. A. Hammond. *Recognizing Free-form Hand-sketched Constraint Network Diagrams by Combining Geometry and Context*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.

[50] T. A. Hammond and B. O'Sullivan. Ladder: a perceptually-based language to simplify sketch recognition user interface development. In *Eurographics Ireland*, pages 67–74, December 2007.

[51] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.

[52] J. I. Hong and J. A. Landay. SATIN: a toolkit for informal ink-based applications. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*, pages 63–72, New York, NY, USA, 2000. ACM.

[53] H. Hse and A. R. Newton. Sketched symbol recognition using zernike moments. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR '04)*, volume 1, pages 367–370, Washington, DC, USA, 2004. IEEE Computer Society.

[54] J. Hu, M. K. Brown, and W. Turin. HMM based on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(10):1039–1045, 1996.

[55] J. Hu, S. G. Lim, and M. K. Brown. Writer independent on-line handwriting recognition using an HMM approach. *Pattern Recognition*, 33(1):133–147, 2000.

[56] T. Igarashi and J. F. Hughes. Clothing manipulation. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*, pages 91–100, New York, NY, USA, 2002. ACM.

[57] T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: a sketching interface for 3D freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, pages 409–416, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[58] P. Isokoski. Performance of menu-augmented soft keyboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*, pages 423–430, New York, NY, USA, 2004. ACM.

[59] J. A. Jorge, M. J. Fonseca, and F. M. G. Pereira. Visual syntax analysis for calligraphic interfaces. In *13° Encontro Portugues de Computacao Grafica*, October 2005.

[60] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, New York, 1972.

[61] J. A. Landay. *Interactive Sketching for the Early Stages of User Interface Design*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, December 1996.

[62] J. A. Landay and B. A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*, pages 43–50, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[63] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *Computer*, 34(3):56–64, 2001.

[64] W. Lee, L. B. Kara, and T. F. Stahovich. An efficient graph-based symbol recognizer. In *Proceedings of the 3rd Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '06)*, pages 11–18, New York, NY, USA, 2006. ACM.

[65] J. Lin, M. W. Newman, J. I. Hong, and J. A. Landay. DENIM: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*, pages 510–517, New York, NY, USA, 2000. ACM.

[66] J. Mankoff and G. D. Abowd. Cirrin: a word-level unistroke keyboard for pen input. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '98)*, pages 213–214, New York, NY, USA, 1998. ACM.

[67] J. Mankoff, G. D. Abowd, and S. E. Hudson. OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers & Graphics*, 24(6):819–834, 2000.

[68] M. Minas. *Spezifikation und Generierung graphischer Diagrammeditoren*. Shaker-Verlag, Aachen, Germany, 2001. Professorial dissertation at Universität Erlangen-Nürnberg, 2000.

[69] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Journal of Science of Computer Programming, Special Issue on Applications of Graph Transformations*, 44(2):157–180, 2002.

[70] M. Minas. Specifying graph-like diagrams with DiaGen. In T. Mens, A. Schürr, and G. Taentzer, editors, *Proceedings of the 1st International Workshop on Graph-Based Tools (GraBaTs '02)*, volume 72 of *Electronic Notes in Theoretical Computer Science*, pages 102–111, Amsterdam, 2002. Elsevier Science Publishers.

[71] M. Minas. Generating meta-model-based freehand editors. In *Electronic Communications of the EASST (GraBaTs '06)*, September 2006.

[72] M. Nakai, T. Sudo, H. Shimodaira, and S. Sagayama. Pen pressure features for writer-independent on-line handwriting recognition based on substroke HMM. In *Proceedings of the 16th International Conference on Pattern*

Recognition (ICPR '02), volume 3, page 30220, Washington, DC, USA, 2002. IEEE Computer Society.

[73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8):12–26, August 1973.

[74] M. Oltmans, C. Alvarado, and R. Davis. ETCHA sketches: Lessons learned from collecting sketch data. In *Making Pen-Based Interaction Intelligent and Natural (AAAI Fall Symposium)*, pages 134–140, Menlo Park, CA, USA, October 2004. AAAI Press.

[75] R. Patel, B. Plimmer, J. Grundy, and R. Ihaka. Ink features for diagram recognition. In *Proceedings of the 4th Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '07)*, pages 131–138, New York, NY, USA, 2007. ACM Press.

[76] B. Paulson and T. Hammond. PaleoSketch: accurate primitive sketch recognition and beautification. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08)*, pages 1–10, New York, NY, USA, 2008. ACM.

[77] B. Paulson, P. Rajan, P. Davalos, R. Gutierrez-Osuna, and T. Hammond. What!?! no rubine features?: using geometric-based features to produce normalized confidence values for sketch recognition. In *Workshop on Sketch Tools for Diagramming (VL/HCC '08)*, pages 57–63, September 2008. https://www.cs.auckland.ac.nz/research/conferences/skekchws/proceedings.html.

[78] K. Perlin. Quikwriting: continuous stylus-based text entry. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*, pages 215–216, New York, NY, USA, 1998. ACM Press.

[79] B. Plimmer and I. Freeman. A toolkit approach to sketched diagram recognition. In *Proceedings of the 21st British HCI Group Annual Conference (HCI '07)*, pages 205–213, September 2007.

[80] B. Plimmer and J. Grundy. Beautifying sketching-based design tool content: issues and experiences. In *Proceedings of the 6th Australasian User Interface Conference (AUIC '05)*, pages 31–38, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[81] P. Rajan and T. Hammond. From paper to machine: Extracting strokes from images for use in sketch recognition. In *Proceedings of the 5th Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '08)*, New York, NY, USA, June 2008. ACM.

[82] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume I: Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[83] D. Rubine. Specifying gestures by example. *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '91)*, 25(4):329–337, 1991.

[84] D. H. Rubine. *The automatic recognition of gestures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[85] E. Saund and E. Lank. Stylus input and editing without prior selection of mode. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*, pages 213–216, New York, NY, USA, 2003. ACM.

[86] P. Schmieder, B. Plimmer, and J. Vanderdonckt. Cross-domain diagram sketch recognition. In *Workshop on Sketch Tools for Diagramming (VL/HCC '08)*, pages 64–73, September 2008. https://www.cs.auckland.ac.nz/research/conferences/skekchws/proceedings.html.

[87] A. Schramm. Übersicht und Klassifizierung von Features für Featurebasierte Erkennungssysteme. Universität der Bundeswehr München, 2008. Bachelor thesis, UniBwM-IS 02/2008, only available in German.

[88] M. Schramm. Vergleich verschiedener Texteingabemöglichkeiten für Sketchingsysteme. Universität der Bundeswehr München, 2008. Bachelor thesis, UniBwM-IS 03/2008, only available in German.

[89] T. M. Sezgin. Feature point detection and curve approximation for early processing of free-hand sketches. Master's thesis, Massachusetts Institute of Technology, Department of EECS, Cambridge, MA, USA, May 2001.

[90] T. M. Sezgin and R. Davis. HMM-based efficient sketch recognition. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI '05)*, pages 281–283, New York, NY, USA, 2005. ACM.

[91] T. M. Sezgin, T. Stahovich, and R. Davis. Sketch based interfaces: early processing for sketch understanding. In *Proceedings of the 2001 Workshop*

on Perceptive User Interfaces (PUI '01), pages 1–8, New York, NY, USA, 2001. ACM.

[92] P. Taele and T. Hammond. Hashigo: A next-generation sketch interactive system for japanese kanji. In *21st Innovative Applications Artificial Intelligence Conference (IAAI '09)*, July 2009.

[93] E. Tapia and R. Rojas. Recognition of on-line handwritten mathematical formulas in the E-Chalk System. In *Proceedings of the 7th International Conference on Document Analysis and Recognition (ICDAR '03)*, pages 980–984, Washington, DC, USA, 2003. IEEE Computer Society.

[94] R. Thierjung. Erkennung und Repräsentation schraffierter und ausgemalter Flächen in Strichzeichnungen. Universität der Bundeswehr München, 2008. Diploma thesis, UniBwM-ID 1/2008, only available in German.

[95] R. Thierjung, F. Brieler, and M. Minas. On-line recognition of hatched and filled regions in hand-drawings. In *Workshop on Sketch Recognition (IUI '09)*, February 2009. http://srl.csdl.tamu.edu/workshops/2009/iui/schedule.html.

[96] M. Thorne, D. Burke, and M. van de Panne. Motion doodles: an interface for sketching character motion. In *Proceedings of the 31st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '04)*, pages 424–431, New York, NY, USA, 2004. ACM.

[97] E. Turquin, M.-P. Cani, and J. F. Hughes. Sketching garments for virtual characters. In *Proceedings of the 1st Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '04)*, pages 175–182, August 2004.

[98] P. A. C. Varley, R. R. Martin, and H. Suzuki. Can machines interpret line drawings? In *Proceedings of the 1st Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '04)*, pages 107–116, 2004.

[99] P. Wais, A. Wolin, and C. Alvarado. Designing a sketch recognition front-end: User perception of interface elements. In *Proceedings of the 4th Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '07)*, pages 99–106, New York, NY, USA, 2007. ACM.

[100] L. Wenyin. On-line graphics recognition: State-of-the-art. In *Proceedings on the 5th International Workshop on Graphics Recognition. Recent Advances and Perspectives (GREC '03)*, volume 3088 of *Lecture Notes in Computer Science*, pages 291–304. Springer, July 2003.

[101] L. Yeung, B. Plimmer, B. Lobb, and D. Elliffe. Levels of formality in diagram presentation. In *Proceedings of the 2007 Conference of the Computer-human Interaction Special Interest Group (CHISIG) of Australia on Computer-human Interaction: Design: Activities, Artifacts and Environments (OZCHI '07)*, pages 311–317, New York, NY, USA, 2007. ACM.

[102] Z. Yuan, H. Pan, and L. Zhang. A novel pen-based flowchart recognition system for programming teaching. In *2nd Workshop on Blended Learning, Revised Selected Papers (WBL '08)*, volume 5328 of *Lecture Notes in Computer Science*, pages 55–64. Springer, August 2008.

[103] R. Zanibbi, D. Blostein, and J. R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(11):1455–1467, November 2002.

[104] S. Zhai and P.-O. Kristensson. Shorthand writing on stylus keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*, pages 97–104, New York, NY, USA, 2003. ACM.

[105] N. Zhu, J. C. Grundy, and J. G. Hosking. Constructing domain-specific design tools with a visual language meta-tool. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE '05), CAiSE Forum*. CEUR-WS.org, June 2005.