INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Visualisierung komplexer reaktiver Systeme
Annotierte Bibliographie

Steffen Prochnow, Reinhard von Hanxleden

Bericht Nr. 0406
21. Juli 2004

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Visualisierung komplexer reaktiver Systeme
Annotierte Bibliographie

Steffen Prochnow, Reinhard von Hanxleden

Bericht Nr. 0406
21. Juli 2004

e-mail: spr@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de

Zusammenfassung

Reaktive Systeme müssen kontinuierlich auf ihre Umwelt reagieren; die meisten eingebetteten Echtzeitsysteme sind reaktive Systeme. Ein wesentliches Hilfsmittel für die Entwicklung reaktiver Systeme ist die *Visualisierung* des Systemverhaltens. Der modellbasierte Entwurf unterstützt diese Verhaltensvisualisierung; bisherige Visualisierungsmechanismen sind jedoch nur für Systeme bis zu einer gewissen Größe effektiv einsetzbar. Dieser Bericht untersucht Ursachen hierfür, und fasst bisherige Ergebnisse auf verschiedenen hierfür relevanten Gebieten der Informatik zusammen.

Inhaltsverzeichnis

1 Einleitung

Reaktive Systeme reagieren fortwährend auf ihre Umgebung; wie reagiert wird, hängt von aktuellen und vorhergegangenen Umgebungsparametern ab. Eingebettete Echtzeitsysteme, weit verbreitet zum Beispiel im modernen Kraftfahrzeug, in der Luftfahrt und in der Medizintechnik, sind generell reaktiv, und stellen hinsichtlich ihrer Komplexität höchste Anforderungen an ihre Entwickler. Dies erfordert robuste, skalierbare Formalismen und Entwicklungsprozesse, um gewünschte Systemeigenschaften erzielen und nachweisen zu können, und unerwünschte Eigenschaften ausschließen zu können. Hierfür ist es erforderlich, auch komplexe Systeme hinsichtlich dieser Eigenschaften für den Entwickler verständlich und nachvollziehbar zu machen. Dies kann wesentlich durch eine *Visualisierung* des Systems unterstützt werden, insbesondere wenn die Visualisierung jeweils an die im Laufe des Entwicklungsprozesses auftretenden Fragestellungen angepasst werden kann. **Die aufgabenspezifische Visualisierung komplexer reaktiver Systeme ist mit den heute verfügbaren Methoden und Werkzeugen jedoch nur eingeschränkt möglich.** Dies ist – nach Meinung der Autoren – inzwischen eine vordringliche Problemstellung, und soll in diesem Bericht näher betrachtet werden. Ein Merkmal dieser Problemstellung ist ihre Querschnittlichkeit durch verschiedene Bereiche der Informatik. Dieser Bericht soll aufzeigen, welche Bereiche der Informatik berührt werden, und welchen Beitrag zur Lösung dieser Problemstellung bisherige Arbeiten auf diesen Gebieten leisten können; dabei wurde die Form der annotierten Bibliographie gewählt.

Der folgende Abschnitt behandelt den modellbasierten Entwurf reaktiver Systeme, insbesondere mit Statecharts. Abschnitt 3 gibt einen Überblick über Verfahren zur Visualisierung großer Informationsmengen, und bewertet diese hinsichtlich ihrer Anwendbarkeit auf die Darstellung des Verhaltens reaktiver Systeme. Abschnitt 4 definiert ästhetische Kriterien für die Visualisierung und fasst bisherige Ergebnisse kognitiver Untersuchungen hierzu zusammen. Abschnitt 5 fasst verfügbare Layoutverfahren zusammen, und bewertet diese hinsichtlich ihrer Anwendbarkeit auf das Layout von Statecharts. Abschnitt 6 beschließt diesen Bericht mit einem Überblick über in diesem Kontext relevante, verfügbare Werkzeuge.

2 Modellierung reaktiver Systeme

Die Komplexität reaktiver Systeme resultiert nicht nur aus der Vielschichtigkeit und Anzahl der Komponenten, also der *Struktur* des gesamten Systems, sondern auch und vor allem aus den Variationsmöglichkeiten und Abhängigkeiten der möglichen Reaktionen des Systems, also seines *Verhaltens* (seiner *Dynamik*). Die Struktur eines Systems lässt sich in der Regel durch sequentielles Vorgehen erfassen, zum Beispiel durch einen *top-down*-Durchgang durch Systemdiagramme. Hierbei wird zunächst die Struktur des Gesamtsystems ohne Details betrachtet, daran anschließend wird das Bild durch rekursives Betrachten der Teilsysteme verfeinert. Hingegen erfordert ein Verstehen des Verhaltens eines reaktiven Systems, dass man über einen gewissen Zeitraum – real oder simuliert – viele Teilsysteme simultan beobachtet. Ein reaktives System umfasst in der Regel viele nebenläufige Aktivitäten, welche einem engen Zusammenspiel unterliegen, und ist damit in seiner Dynamik generell schwieriger zu erfassen als ein klassisches, sequentiell ablaufendes Programm, welches nur einen einzelnen *locus of control* hat. Ein inzwischen etablierter Ansatz für den Entwurf und die Analyse reaktiver Systeme ist die Erstellung eines ausführbaren, graphischen Systemmodells. Hiermit kann das Systemverhalten simuliert und – je nach Modellierungsumgebung – formal verifiziert werden, bevor ein konkreter Prototyp erstellt und getestet wird. Hierbei kann die formale Verifikation dazu dienen, gezielte Fragestellungen zum Systemverhalten zu beantworten, zum Beispiel hinsichtlich der Erreichbarkeit bestimmter Zustände; für ein generelles Verständnis des Systemverhaltens bleibt die explorative Simulation und Visualisierung des Systemverhaltens jedoch unerlässlich. Es existieren bereits akzeptierte Paradigmen und Werkzeuge zur Verhaltensvisualisierung, wie zum Beispiel Statecharts; jedoch sind diese hinsichtlich der angebotenen (starren) Sichten und ihrer Bedienbarkeit

nicht skalierbar für komplexe Systeme.

2.1 Statecharts

Der modellbasierte Entwurf reaktiver Systeme [8] ist seit längerem ein eigenständiges Forschungsthema in der Informatik. Ein inzwischen weitreichend untersuchter und durch eine Reihe kommerzieller und am Markt erfolgreicher Werkzeuge unterstützter Formalismus für das Beschreiben reaktiven Verhaltens beruht auf Statecharts [6]. Diese erweitern endliche Automaten [1] um Nebenläufigkeit und Hierarchie, und können damit auch komplexe Verhaltensmuster kompakt und strukturiert beschreiben. Statecharts sind ein graphischer, syntaktischer Formalismus, der inzwischen auch Teil der *Unified Modeling Language* (UML [9]) ist. Eine Vielzahl von Arbeiten haben sich mit möglichen Semantiken und darauf aufbauenden formalen Analysen von Statecharts befasst [2, 5, 4, 7, 3].

Ein wesentlicher Vorteil von Statecharts ist deren Ausführbarkeit; Statecharts unterstützende Modellierungswerkzeuge bieten generell nicht nur einen Statechart-Editor, sondern auch die Möglichkeit zur Simulation und zur Codesynthese. Ein gängiges, seit den Anfängen der Statecharts unverändertes Paradigma zur Visualisierung des Systemverhaltens während eines Simulationslaufs ist die Hervorhebung der jeweils aktiven Teilzustände und Transitionen. Eine Einschränkung dieses Ansatzes ist, dass die Präsentation des Statecharts – von diesen farblichen Hervorhebungen abgesehen – auch während der Simulation unverändert bleibt, und zwar mit dem Detaillierungsgrad und dem Layout, welche der Modellierer beim Erstellen des Statecharts vorgegeben hat. Einzelne Statecharts lassen sich so gut nachvollziehen; diese spiegeln im Allgemeinen jedoch jeweils nur einzelne Komponenten eines Systems wieder. Zum Teil bieten Werkzeuge, wie das in Abschnitt 6 näher beschriebene ArgoUML [10], als Ergänzung noch komprimierte, tabellarische Übersichten zu Statechart-Elementen, dies jedoch während einer Simulation nicht dynamisch angepasst.

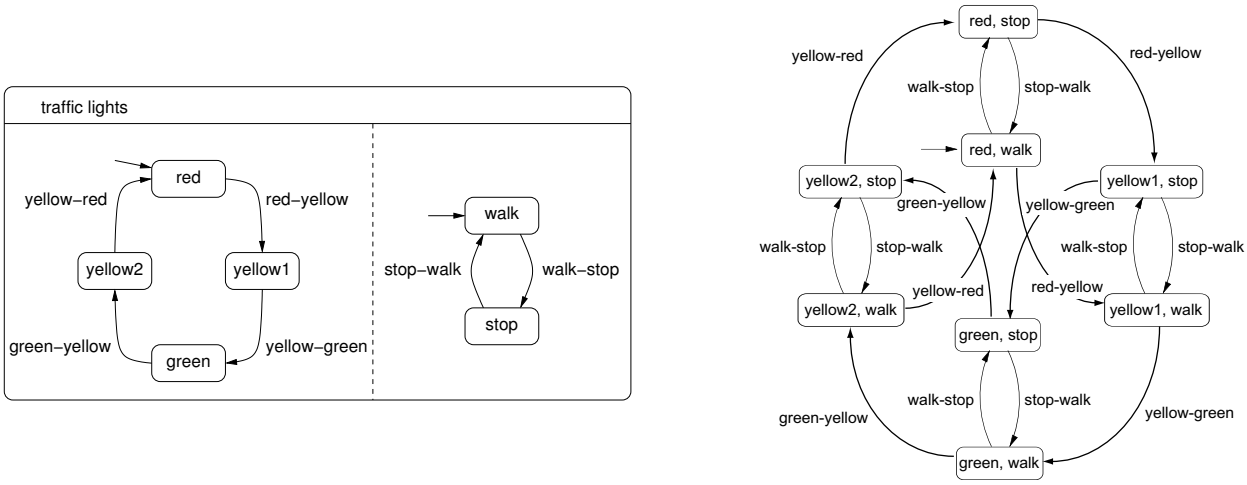
Modellierungswerkzeuge erlauben generell die Selektion einer Menge von Statecharts, welche dann jeweils in einem eigenen Fenster angezeigt werden und beliebig am Bildschirm angeordnet werden können. Für realistische Systeme ist es jedoch oft schwierig, die gerade „aktiven“, für das Verständnis wesentlichen Statecharts gleichzeitig sichtbar zu haben; weiterhin ändert sich diese Menge oft im Laufe einer Simulation. Konkret bedeutet dies aus Anwendersicht ein zeitaufwändiges und von der eigentlichen Aufgabe ablenkendes Selektieren und Verschieben von Fenstern, und bei nicht ausreichender Darstellungsfläche zusätzlichen mentalen Integrationsaufwand.

2.2 Systemsichten

Statecharts scheinen ein inzwischen gut verstandenes, bewährtes Instrument zur *Definition* und *Dokumentation* der Dynamik einzelner Systemkomponenten und damit auch von Gesamtsystemen. Jedoch schränkt die mit Statecharts klassischerweise assoziierte, statische Sichtweise deren Eignung zur *Analyse* reaktiven Verhaltens stark ein, wie wiederum die eigenen Erfahrungen der Autoren beim Entwurf industrieller Anwendungen gezeigt haben. Das zugrundeliegende Problem ist die mit bisherigen Visualisierungsansätzen zu geringe Informationsdichte [12] von Statecharts bezüglich der während der Simulation relevanten Informationen. Dies hat zur die Konsequenz, dass sich jeweils nur kleine Teile des Systemzustands sichtbar machen lassen, auch wenn z. B. eine textuelle Codierung der jeweiligen Systemkonfiguration bequem auf einer Schreibmaschinenseite Platz finden würde. Eine solche textuelle Darstellung hat jedoch den Nachteil, dass der Bezug zum Systemmodell nicht mehr offensichtlich ist, und der Vorteil der visuellen Erlebbarkeit des dynamischen Systemverhaltens verloren geht.

Das Nachvollziehen der Dynamik eines Systems kann in zwei Fragestellungen unterteilt werden:

1. „*Wie* verhält sich das System?“
2. „*Warum* verhält sich das System so wie es sich verhält?“



(a) (b)

Abbildung 1: Eine Ampelsteuerung als paralleles Statechart (a) und das entsprechende kartesische Produkt (b) (letzteres entnommen von Berard et al. [14]).

Für die Beantwortung der ersten Frage ist es oft nicht vorrangig, die Details eines Modells zu sehen, insbesondere für Anwender, welche das Modell selbst erstellt haben und damit bereits relativ vertraut sind; wichtiger ist generell, einen Überblick über das gesamte System zu haben. Erst für die zweite Frage wird es in der Regel erforderlich, detailliertere Informationen zu einzelnen Teilsystemen zu bekommen. Die Aufgabe ist also, eine große Menge an (statischen) Informationen über die Zustandslogik einzelner Systemkomponenten sinnvoll zu reduzieren, und diese mit möglichst umfassenden Informationen über die (dynamischen) Systemzustände zu kombinieren.

Im Folgenden unterscheiden wir zwischen *statischen Sichten* auf ein Systemmodell, welche sich nicht auf eine konkrete Systemkonfiguration beziehen, und *dynamischen Sichten*, welche sich auf eine konkrete Systemkonfiguration beziehen, welcher z. B. im Laufe einer Simulation erreicht worden ist. Eine Abfolge von dynamischen Sichten ist eine *Animation* eines tatsächlichen Systemverhaltens. Orthogonal dazu unterscheiden wir zwischen *starren Sichten*, welche für ein gegebenes Teilsystem stets die gleichen Informationen in der gleichen Art und Weise präsentieren, und *flexiblen Sichten*, welche hinsichtlich Auswahl und Darstellung der dargebotenen Informationen variieren können. Weiterhin unterscheiden wir zwischen *lokalen Sichten*, welche Systemausschnitte isoliert ohne Kontext und Querbezüge umfassen, und *globalen Sichten*, welche das Gesamtsystem umfassen. Herkömmliche Visualisierungsparadigmen beschränken sich auf lokale, starre Sichten.

2.3 Die Darstellung von Produktautomaten

Statecharts erlauben Parallelität, was deren Mächtigkeit gegenüber einfachen endlichen Automaten drastisch erhöht, da hierdurch Redundanzen und das einhergehende Problem der Zustandsexplosion vermieden werden kann. Statecharts lassen sich jedoch durch Produktbildung in einfache, „flache“ Automaten überführen. Abbildung 1 zeigt als Beispiel den Produktautomaten von zwei parallelen Ampelsteuerungen. Falls der Produktautomat nicht zu groß ist, kann dies eine hilfreiche alternative Darstellung sein. Insbesondere ließe sich am Produktautomaten der Verlauf einer Simulation besser nachvollziehen als am parallelen Statechart. So würde zum Beispiel für die Ampelsteuerung eine farbliche Markierung der im Laufe einer Simulation erreichten Zustände im Produktautomat unmittelbar verdeutlichen, ob eine verbotene Konfiguration („green, walk“) erreicht worden ist. Dies ließe sich aus einer entsprechenden Markierung des ursprünglichen, parallelen Statecharts nicht ablesen, da auch bei einer korrekten Ampelsteuerung sowohl der Zustand „green“ als auch der Zustand

„walk“ erreicht werden könnte, nur nicht im selben Simulationsschritt.

Für Produktautomate wird zunächst die Reihenfolge der Teilautomaten bestimmt, die man mit $a_1, \ldots, a_n$ bezeichnet. Das Statechart von a_1 wird für jeden Zustand von a_2 einmal dargestellt, wobei die einzelnen Darstellungen von a_1 wie die Zustände von a_2 angeordnet werden. Dieses Prinzip wird für alle a_i fortgesetzt. Zum Schluss werden die Zustandsbezeichnungen vervollständigt und die Transitionen eingefügt. Zur Darstellung wäre auch eine mehrdimensionale Darstellung geeignet.

Frank van Ham et al. haben ein Verfahren zur Visualisierung großer Zustandsräume entwickelt, welches Zustände in *cluster* zusammenfasst, wenn diese die gleiche Distanz vom Ursprungszustand haben, und deren Nachfolgezustandsmengen nicht disjunkt sind [13]. Die hieraus resultierenden Baumstrukturen werden dreidimensional als *cone trees* [11] visualisiert. Sie haben dieses Verfahren auf Kommunikationsprotokolle mit sehr großen Zustandsräumen angewandt, und als effektive Ergänzung zu formalen Methoden gesehen, um Einblicke in das Verhalten der Protokolle zu erlangen. Sie betrachten stets „flache“ Zustandsräume, ohne hierarchische Strukturierung; diese Verfahren ließen sich also auch auf Produktautomaten anwenden.

Literatur

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Michael von der Beeck. A comparison of statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

[3] Tom Bienmüller, Werner Damm, and Hartmut Wittke. The statemate verification environment – making it real. In E. Allen Emerson and A. Prasad Sistla, editors, *12th international Conference on Computer Aided Verification CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 561–567. Springer, 2000.

[4] William Chan, Richard J. Anderson, Paul Beame, David H. Jones, David Notkin, and William E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, February 2001.

[5] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In A. Pnueli W.-P. de Roever, H. Langmaack, editor, *Compositionality: The Significant Difference: International Symposium, COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*, pages 186–238. Springer, 1998.

[6] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] Jan Jürjens. A UML statecharts semantics with message-passing. In *Symposium of Applied Computing (SAC 2002), Madrid*, 2002.

[8] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE Software*, 20(5):14–18, September/October 2003.

[9] Object Management Group. Omg unified modeling language specification, version 1.5, March 2003. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`.

[10] Jason Robbins and David Redmiles. Cognitive support, uml adherence, and xmi interchange in argo/uml. *Journal of Information and Software Technology. Special issue: The Best of COSET '99*, 42(2):79–89, 2000.

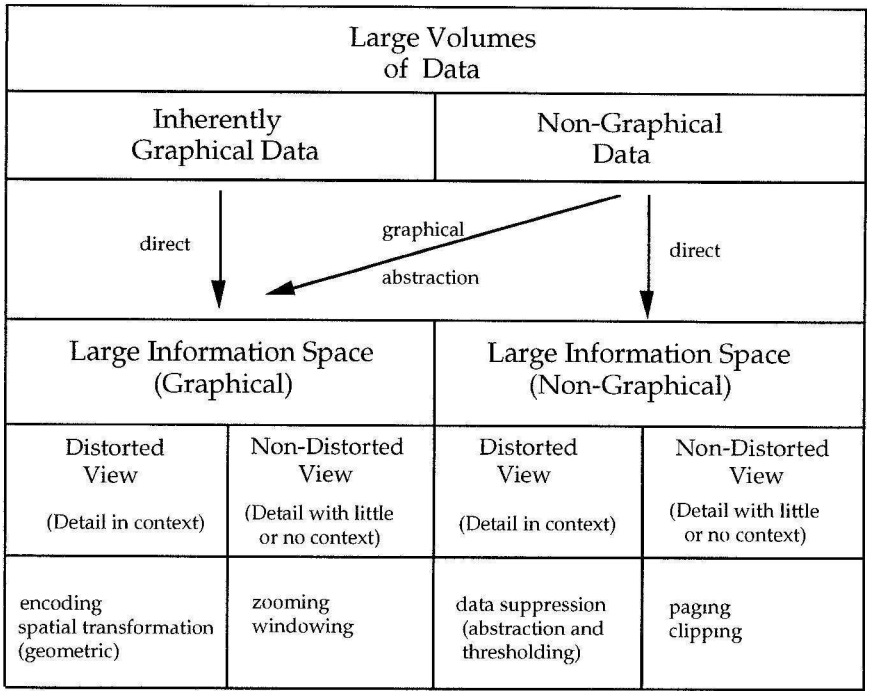


Abbildung 2: Klassifizierung von Präsentationstechniken großer visueller Informationen nach Leung und Apperley [19].

[11] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM Press, 1991.

[12] Edward R. Tufte. *Visual Explanations.* Graphics Press, Cheshire, Connecticut, 1997.

[13] Frank van Ham, Huub van de Wetering, and Jarke J. van Wijk. Interactive visualization of state transition systems. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):319–329, October/December 2002.

3 Die Visualisierung großer Informationsmengen

Statecharts mit einer gewissen Ausdehnung können nicht auf einer begrenzten Fläche dargestellt werden, ohne dass dem Betrachter Statechart-Teile oder Details verborgen bleiben. Es existiert eine Reihe von Ansätzen für dynamische Darstellungen, von denen einige im Folgenden kurz vorgestellt und hinsichtlich ihrer Eignung für Statechart-Darstellungen untersucht werden. Es werden zwei Ansätze zur Visualisierung unterschieden:

Zoom: optisches oder graphisches Vergrößern von Ausschnitten relevanter Bereiche

Ausklappen oder Explodieren: Veränderung der Sichtbarkeit von Unterzuständen von Hierarchiezuständen

Abbildung 2 zeigt eine allgemeine Übersicht zur Klassifizierung von Darstellungsalgorithmen von visuellen Informationen.

3.1 Konventionelle Zoomable User Interfaces

Die bekannteste und aufgrund ihrer Einfachheit am häufigsten implementierte Technik zur Visualisierung von Diagrammen ist die Kombination von Skalierung (*Zooming*) und Positionierung (*Panning*). Die Darstellung wird auf einen rechteckigen Ausschnitt beschränkt, dessen Größe und Position relativ zum Diagramm mit Hilfe von Zoom-Werkzeugen, Scrollbars und ähnlichen Bedienelementen festgelegt wird. Systeme, die diese Technik verwenden, werden *Zoomable User Interfaces* genannt und leiden unter folgenden Defiziten:

- Durch die Beschränkung der Darstellung auf einen Ausschnitt, gehen die Kontextinformationen verloren. Der Nutzer sieht nicht unmittelbar, welcher Teil des Diagramms angezeigt wird und wie dieser Teil mit anderen Bereichen in Verbindung steht.
- Je größer der dargestellte Ausschnitt ist, desto mehr Details werden angezeigt. Die Erweiterung des sichtbaren Bereiches beeinträchtigt die Übersichtlichkeit; gleichzeitig werden die einzelnen Elemente verkleinert und können schlechter wahrgenommen werden.
- Schon nach wenigen Navigationsschritten weiß der Nutzer oft nicht mehr, an welchem Ort des Informationsraumes er sich befindet und auf welchem Weg er die gesuchte Information finden kann.

3.2 Mehrere Ansichten in unterschiedlichen Abstraktionsstufen

Dem Nutzer kann die Möglichkeit gegeben werden, mehrere Ansichten in unterschiedlichen Vergrößerungsstufen parallel zu verwenden. Ein Spezialfall dieses Prinzips ist die Präsentation einer weniger detaillierten Überblickansicht in einem separaten Bildschirmbereich oder alternativ zur Arbeitsansicht. Die Überblicksansicht kann auch als dauerhafte (*Overview-Layer* [15], Abb. 3) oder temporäre (*Context-Layer* [21]) transparente Ebene über der Arbeitsansicht angezeigt werden. Der aktuell gewählte Ausschnitt wird normalerweise durch eine rechteckige Umrandung in der Überblicksansicht markiert und kann durch Verschieben dieses Rechtecks geändert werden. Um die Auswahl des Bildausschnitts noch flexibler zu gestalten, kann es dem Nutzer ermöglicht werden, mit der Maus einen rechteckigen Bereich beliebiger Größe in der Übersichtsansicht oder der aktuellen Arbeitsansicht zu wählen. Die Übersichtsansicht muss als Abstraktionsmethode nicht zwangsläufig die optische Verkleinerung verwenden. Oft bietet es sich an, alternative Navigationsstrukturen zur Verfügung zu stellen, indem die präsentierten Informationen anders organisiert werden. Ein Beispiel dafür sind Baumdarstellungen für hierarchisch strukturierte Informationen, wie sie in vielen *CASE*-Werkzeugen verwendet werden. Die Kombination aus Arbeits- und Übersichtsansicht den Nachteil, dass mehr Platz auf dem Bildschirm benötigt wird und der Nutzer gezwungen ist, beide Ansichten mental zu integrieren.

3.3 Die Animation von Navigationsvorgängen

Pook et. al. [21] stellen eine weitere Navigationshilfe vor, die zur Veranschaulichung des gegenwärtigen Kontextes beitragen kann. Durchgeführte Navigationsschritte, z. B. Vergrößerungen, Verkleinerungen und Verschiebungen des Bildausschnitts, werden auf Verlangen des Nutzers im Schnelldurchlauf wiederholt, wobei die jeweilige Ansicht in einer transparenten Ebene (*History-Layer*) über der Arbeitsansicht dargestellt wird. Auf diese Weise soll dem Anwender in Erinnerung gerufen werden, auf welchem Weg der aktuelle Bildausschnitt gewählt wurde. Das Verfahren lässt sich insofern vereinfachen, dass anstatt der vom Nutzer durchgeführten Navigationsschritte ein direkter Übergang von der Gesamtsicht zum aktuellen Bildausschnitt präsentiert wird; diese Variante wirkt oft weniger verwirrend.

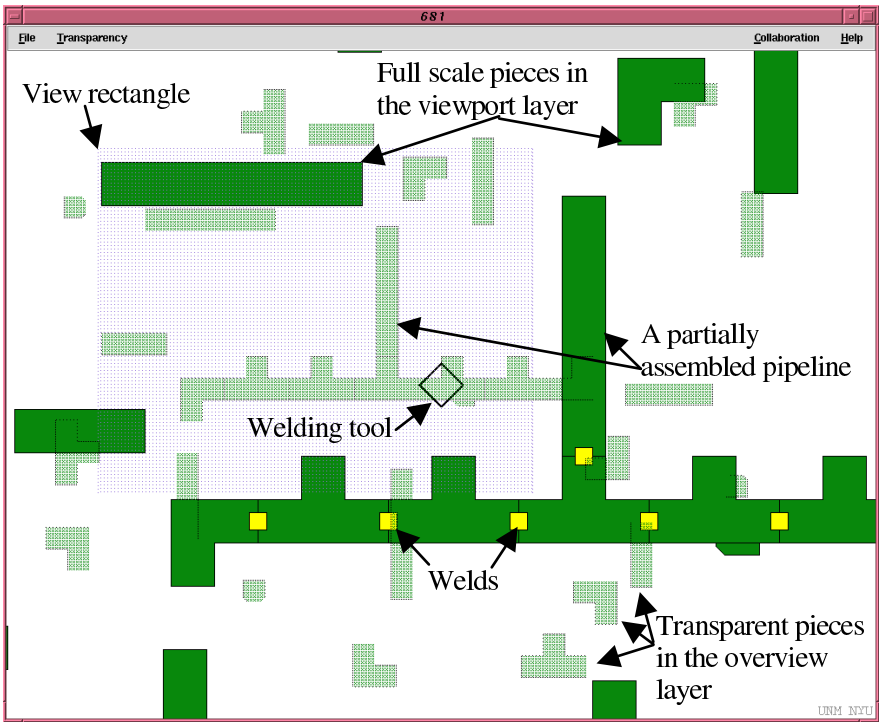


Abbildung 3: Overview-Layer nach Cox et al. [15].

3.4 Fokus-und-Kontext-Darstellung

Die Fokus-und-Kontext-Darstellung soll es ermöglichen, die für den Nutzer besonders interessanten Informationen detailliert darzustellen und trotzdem den Bezug zur Umgebung zu erhalten. Der Fokus wird in einer hohen Vergrößerungsstufe präsentiert, so dass alle Einzelheiten erkennbar sind. Für den Kontext wird eine geringere Vergrößerung gewählt, um eine möglichst große Informationsmenge auf kleinerem Raum darstellen zu können. Die wahrscheinlich bekannteste Realisierung der Fokus-und-Kontext-Darstellung ist das *Fisheye View*. Mit Hilfe einer nichtlinearen optischen Verzerrung wird ein meist kreisförmiger Bereich um den Fokus vergrößert. Das Ergebnis entspricht in etwa dem Blick durch eine auf die Darstellung gelegte Sammellinse oder einer fotographischen Aufnahme mit einem Weitwinkelobjektiv. Die linke Seite der Abb. 5 zeigt ein Diagramm in Fisheye-Darstellung.

In der Literatur werden zwei wichtige Nachteile der Fisheye-Darstellung genannt:

- Statecharts bestehen zum Großteil aus geometrischen Formen und Text; ihre Lesbarkeit wird durch die optische Verzerrung erschwert
- Der hohe Rechenaufwand könnte die Darstellung neuberechneter Teile des Statecharts verzögern und somit die Nachvollziehbarkeit erschweren.

Sarkar und Brown [22] stellen eine alternative Fokus-und-Kontext-Darstellung vor, die nicht auf optischen Verzerrungen beruht. Die entstehenden Diagramme sollen trotz der Vergrößerung des Fokusbereiches ihren natürlichen Charakter behalten. Position, Größe und Detaillierungsgrad der Objekte können anwendungsspezifisch berechnet werden.

Begriffe:

Degree of Interest (DOI): numerischer Wert, setzt sich zusammen aus

Level of Detail (LOD): Wichtigkeit des Objektes *a priori*, ist unabhängig von der aktuellen Wahl des Bildausschnittes

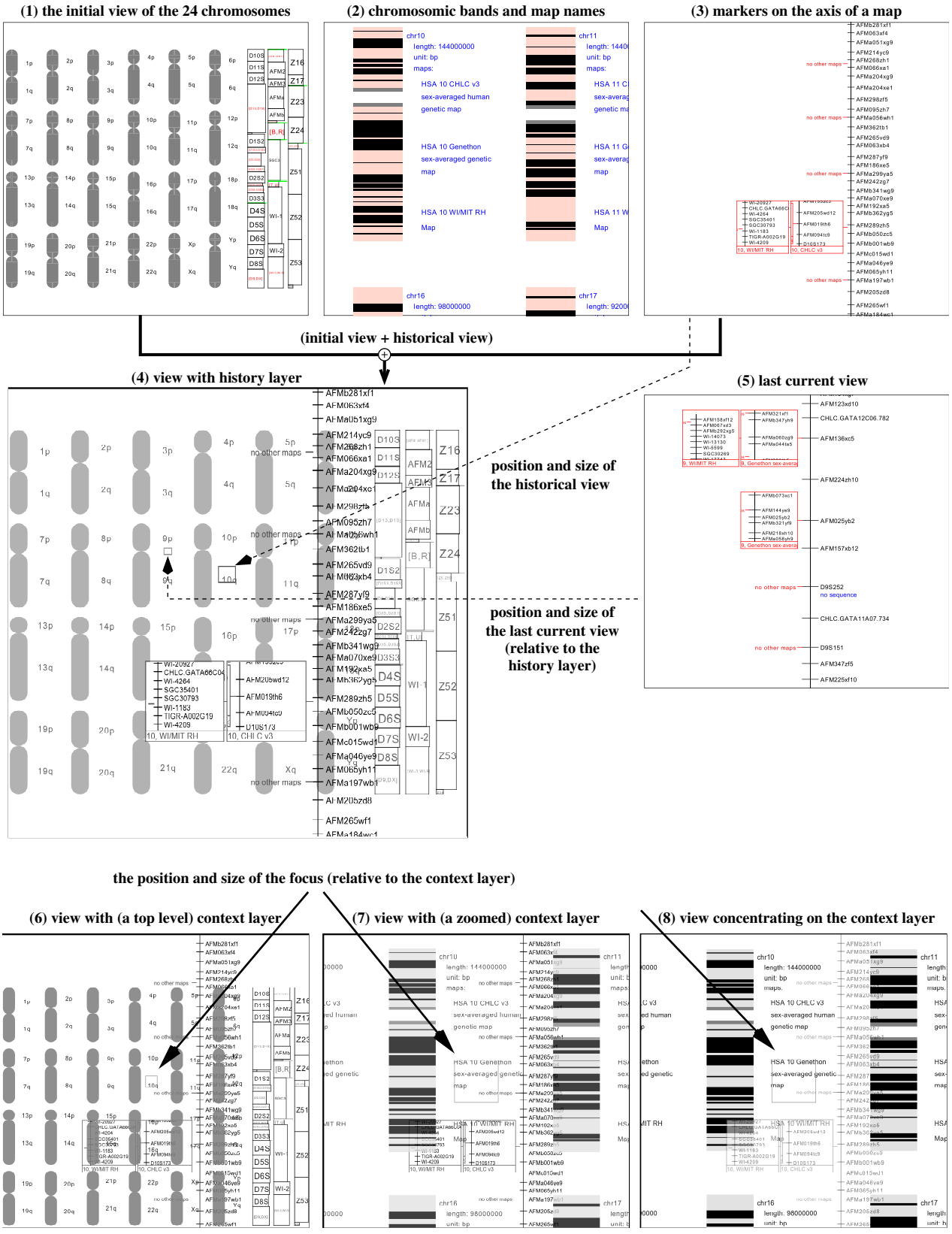


Abbildung 4: Context-, History und Overview-Layer nach Pook et al. [21].

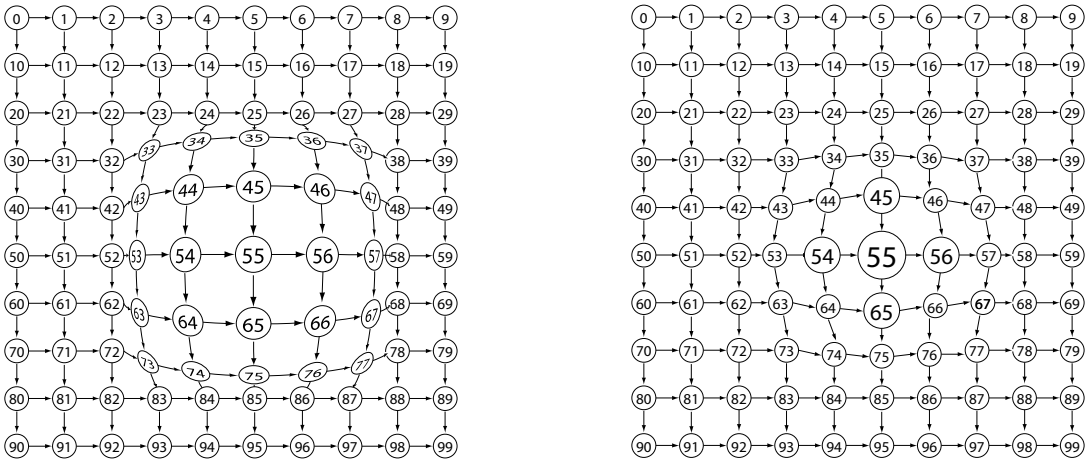


Abbildung 5: Diagramm in optischer (links) und graphischer (rechts) Fokus-und-Kontext-Darstellung.

Distance from Focus (DFF): Abstand des Objektes vom Zentrum des Benutzerinteresses.

Ein Objekt wird um so größer dargestellt, je größer es von Natur aus ist und je näher es dem Zentrum der Aufmerksamkeit liegt.

3.5 Semantisches Zooming

Eine Möglichkeit, ein Statechart in verschiedenen Vergrößerungsstufen sinnvoll zu präsentieren, ist das *semantische Zooming*. Dabei wird der Detaillierungsgrad der dargestellten Informationen an die jeweilige Vergrößerungsstufe angepasst. Wenn der sichtbare Bereich viele Elemente enthält, werden diese auf einem entsprechend hohen Abstraktionsniveau angezeigt, indem unwichtige Informationen ausgeblendet werden. Auf diese Weise lässt sich eine annähernd konstante optische Komplexität über mehrere Vergrößerungsstufen hinweg erreichen. Das semantische Zooming basiert auf einer hierarchischen Strukturierung der darzustellenden Elemente. Mit abnehmender Vergrößerungsstufe werden mehr Hierarchieebenen ausgeblendet. Um bei jeder Vergrößerung sinnvolle und ästhetische Darstellungen zu erhalten, müssen Form und Lage der Objekte gegebenenfalls angepasst werden. Dazu ist es nötig, die Objekte in Zusammenhang mit ihrem Kontext zu betrachten.

3.6 Semantische Fokus-und-Kontext-Darstellung

Koeth [17] stellt eine Kombination von Fokus-und-Kontext-Darstellung und semantischem Zooming vor, die er als semantische Fokus-und-Kontext-Darstellung (sFK) bezeichnet. Fokus und Kontext werden in einer unterschiedlichen Abstraktionsstufe dargestellt; der geringere Abstraktionsgrad des Kontexts wird jedoch nicht durch optische Verkleinerung, sondern durch Ausblenden von Details im Sinne des semantischen Zooming realisiert. Für die Verwendung der semantischen Fokus-und-Kontext-Darstellung spricht die Tatsache, dass Diagramme jederzeit in einer syntaktisch korrekten und verzerrungsfreien Form präsentiert werden können.

3.7 Das Ausblenden nicht erreichbarer Zustände

Insbesondere durch die Bildung von Produktautomaten (siehe Abschnitt 2.3) können Diagramme mit einer großen Anzahl graphischer Objekte entstehen. Wenn alle nicht erreichbaren Zustände auf Wunsch ausgeblendet werden können, ließe sich die Übersichtlichkeit verbessern. Außerdem könnte der direkte Vergleich zwischen erreichbarer und gesamter Zustandsmenge den Nutzer unter Umständen bei der Modellierung unterstützen.

Literatur

[14] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Beatrice Berard, Michel Bidoit, Alain Finkel, Francois Laroussiniea, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer-Verlag, Berlin Heidelberg, 2001.

[15] D. Cox, J. S. Chugh, C. Gutwin, and S. Greenberg. The Usability of Transparent Overview Layers. In *Companion Proceedings of the CHI '98 Conference on Human Factors in Computing Systems*. ACM Press, 1997.

[16] Gerge W. Furnas. Generalized Fisheye Views. In *Human Factors in Computings Systems CHI '86 Conference Proceedings*, pages 16–23, Bell Communications Research, 435 South St., Morristoen, New Jersey, 1986.

> Annotation: This paper explores fisheye views presenting, in turn, naturalistic studies, a general formalism, a specific instantiation, a resulting computer program, example displays and an evaluation.

[17] Oliver Köth. Semantisches Zoomen in Diagrammeditoren am Beispiel von UML. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001.

> Annotation: This work deals with presentation and handling of large diagrams in graphical editors, which were produced by the DiaGen system. It is shown that work with large graphical representations requires particular support by the editor to obtain lucidity and manageability.

[18] Oliver Köth and Mark Minas. Structure, Abstraction and Direct Manipulation in Diagram Editors. In M. Hegarty et. al., editor, *LNAI 2317*. Springer Verlag, 2001.

[19] Y. K. Leung and M. D. Apperley. A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, June 1994.

[20] J. D. Mackinlay, G. G. Robertson, and S. K. Card. The Perspective Wall: Detail and context smoothly integrated. In *Proc. of CHI '91*, New Orleans, April 1991.

> Annotation: This paper describes a technique called the *Perspective Wall* for visualizing linear information by smoothly integrating detailed and contextual views. The resulting visualizations supports efficient use of space and time.

[21] Sutoart Pook, Eric Lecolinet, Guy Vayssaix, and Emmanuel Barillot. *Context and Interaction in Zoomable User Interfaces*. ACM Press, 2000.

> Annotation: This paper proposes a temporary in-place context aid that helps users position themselves in Zoomable User Interfaces (ZUIs). This context layer is a transparent view of the context that is drawn over the users focus of attention. A second temporary in-place aid is proposed that can be used to view already visited regions of the information space. This history layer is an overlapping transparent layer that adds a history mechanism to ZUIs. These orientation aids are completed with an additional window, a hierarchy tree, that shows users the structure of the information space and their current position within it. Context layers show users their position, history layers show them how they got there, and hierarchy trees show what information is available and where it is.

[22] Manojit Sarkar and Marc H. Brown. Graphical Fisheye Views of Graphs. In *Proceedings of the ACM SIGCHI 1992 Conference on Human Factors in Computing Systems*, 1992.

> Annotation: This paper describes a system for viewing and browsing graphs using a software analog of a fisheye lens. It first shows how to implement such a view using solely geometric transformations. Then it describes a more general transformation that allows hierarchical or structured information about the graph to affect the view. The general transformation is a fundamental extension to previous research in fisheye views.

4 Ästhetische Kriterien, kognitive Untersuchungen und Layout-Konventionen

Eine wesentliche Fragestellung bei der Darstellung von Statecharts ist, welches geeignete „ästhetische Kriterien“ für das Layout statischer und dynamischer Statecharts sind. Für allgemeine Graphen gibt es bereits eine Reihe von Arbeiten. Davidson und Harel [28] kombinieren eine Reihe ästhetischer Kriterien, welche die Güte eines Graphen aus Sicht des menschlichen Betrachters quantifizieren. Sie schlagen einen Layoutalgorithmus vor, welcher versucht, die daraus resultierende Gütefunktion zu optimieren. Coleman und Parker [27] verfolgen einen ähnlichen Ansatz, und präsentieren den gegenüber Davidson und Harel optimierten AGLO (*Aesthetic Graph Layout*) Algorithmus. Dabei werden jeweils einfache Graphen mit punktförmigen Knoten und geraden Kanten angenommen. Sie unterscheiden dabei zwischen *syntaktischer Ästhetik* („Knoten sollten nicht zu nahe aneinander platziert werden“) und *semantischer Ästhetik* („Knoten eines Aufruf-Graphen, welche Funktionen innerhalb des gleichen Moduls entsprechen, sollten nahe aneinander platziert werden“). Außerdem differenzieren sie zwischen *statischer Ästhetik* und *dynamischer Ästhetik*, letztere für den Fall, dass ein Graph Modifikationen unterliegt. Ein wesentliches dynamisches Ästhetik-Kriterium ist dabei, dass sich die Positionen bestehender Knoten und Kanten bei Modifikationen möglichst wenig verändern sollten (*minimal edit disruption*); dies wird auch als *dynamische Stabilität* bezeichnet. Damit soll erreicht werden, dass eine vom Benutzer bereits verinnerlichte Struktur, die *kognitive Karte* (*mental map*), bei Bearbeitungen möglichst wenig verändert werden muss; hierzu existieren bereits eine Reihe von Ansätzen [44, 45]. Kosak et al. [31] unterscheiden hingegen *syntaktische Korrektheit*, *perzeptuellen Aufbau*, und *ästhetische Optimalität* eines Graphen, vergleiche Abbildung 6. Der perzeptuelle Aufbau eines Graphen lässt sich dabei, ähnlich wie die semantische Ästhetik, auch als *Sekundärnotation* (siehe Abschnitt 4.2) nutzen, um den Informationsgehalt eines Graphen zu erhöhen und dessen Lesbarkeit zu verbessern.

4.1 Normalformen

Semantische Ästhetikkriterien haben jeweils lokalen Charakter; sie können jedoch zu *Normalformen* erweitert werden, welche sich generell auf das gesamte Statechart bezieht. Normalformen haben präskriptiven, normativen Charakter und verfolgen das Ziel, eine Vielfalt von Möglichkeiten der Darstellung eines Statecharts auf eine einzige zu reduzieren, um zum Beispiel den Vergleich zwischen Statecharts zu erleichtern – wohingegen semantische Ästhetikkriterien prinzipiell mehrere gleich gute Lösungen zulassen. Wir sehen daher eine Normalform als eine Kombination von *normativen Ästhetikkriterien*, wie zum Beispiel

- „Der initiale Zustand ist jeweils links oben“;
- „Transitionen von links nach rechts verlaufen oberhalb von Quelle und Senke“;
- „Transitionen verlaufen im Uhrzeigersinn“.

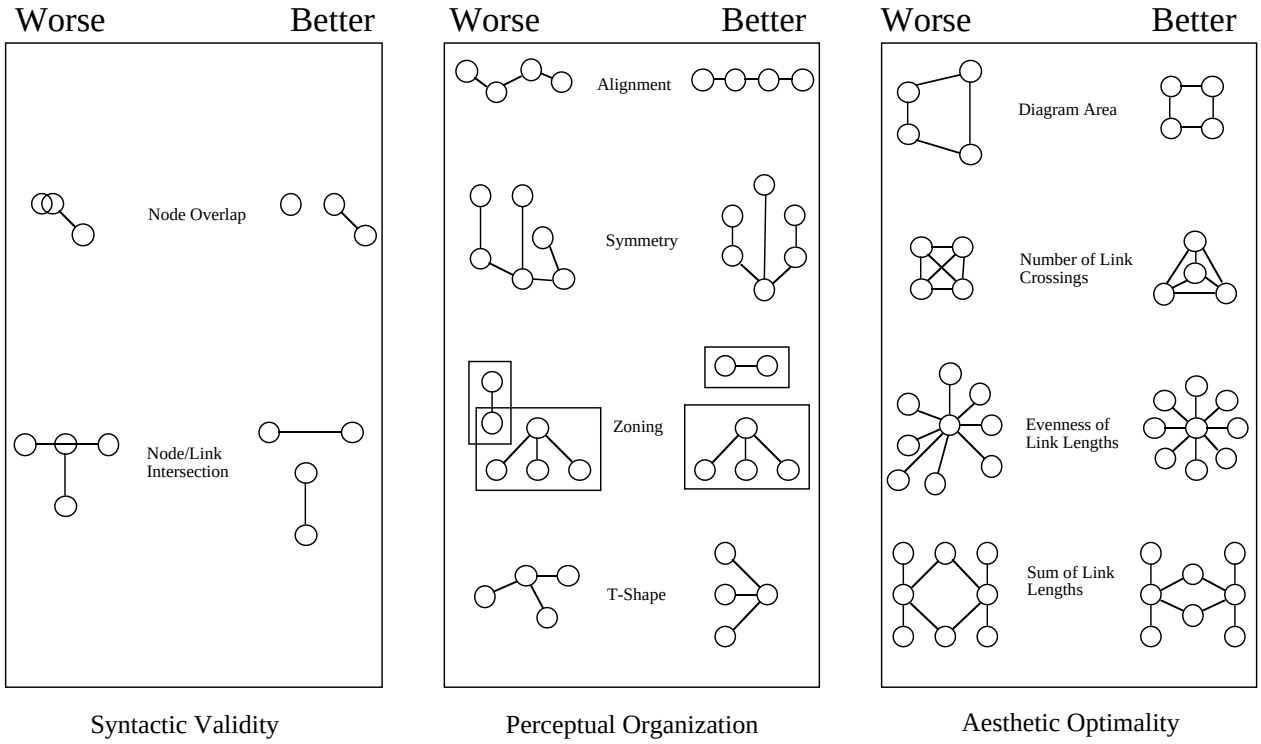


Abbildung 6: Unterscheidung von syntaktischer Korrektheit, perzeptuellem Aufbau, und ästhetischer Optimalität nach Kosak et al. [31].

Ein Indiz für das Potenzial von Normalformen und normativen Ästhetikkriterien sind die schon mit einem einfachenHorizontal/Vertikal-Layouter gemachten Erfahrungen [30]. So führt zum Beispiel die konsequente Anordnung des initialen Zustands links bzw. oben dazu, dass ein geübter Anwender nicht mehr nach der initialen Transition suchen muss, um diese zu identifizieren. Weiterhin führt die konsequente Anordnung von Transitionen im Uhrzeigersinn dazu, dass nicht mehr die Pfeilspitze lokalisiert werden muss, um die Richtung der Transitionen abzuleiten. Dies ist insbesondere bei kondensierten Darstellungen wertvoll und macht zum Beispiel offensichtlich, welche Zustände viele mögliche Vorgänger- bzw. Nachfolgezustände haben. Weiterhin liefert eine lineare Anordnung der Zustände mit minimalen Transitionskreuzungen einen schnellen, qualitativen Überblick über den Fortschritt einer Simulation.

Schließlich bieten normative Ästhetikkriterien neben den kognitiven Vorteilen in der Regel auch Vorteile hinsichtlich der für ein automatisches Layout erforderlichen Rechenzeit, und möglicherweise auch für den erforderlichen Speicheraufwand; eine lineare Anordnung lässt sich generell schneller berechnen und kompakter darstellen als eine beliebige zweidimensionale Anordnung.

4.2 Kognitive Untersuchungen zu visuellen Sprachen

„Ein Bild sagt mehr als Tausend Worte“ – ist eine verkürzte Klassifizierung und Bewertung von alternativen Möglichkeiten, das Gleiche auszudrücken. Der Mensch ist ein „visuelles Wesen“ und kann Informationen schneller graphisch aufnehmen als über das geschriebene oder gesprochene Wort. In der Informatik kann diese Klassifizierung auf Sprachebenen von der Spezifikation und Modellierung bis zur Programmierung angewandt werden. Beispiele „bildhafter“, also graphischer, visueller Sprachen sind Flussdiagramme, Klassendiagramme und Petrinetze. Sehr häufig bieten die Werkzeuge auch die Möglichkeit, aus dieser graphischen Spezifikationssprache ein (textuelles) Programm z. B. in C oder Ada zu generieren. Im Folgenden werden *graphische Sprachen* und *visuelle Sprachen* syn-

onym verwendet; im Gegensatz dazu stehen *textuelle Sprachen*. Ein *Diagramm* ist eine mit einer visuellen Sprache erstellte Beschreibung, bzw. ein Fragment davon; analog dazu ist *Text* eine mit einer textuellen Sprache erstellte Beschreibung.

Visuelle Sprachen werden generell mit einer Reihe von Vorteilen assoziiert, unter anderem:

- Komplexe Strukturen können relativ schnell erfasst werden;
- Es gibt relativ wenig Sprachelemente, und diese lassen sich relativ schnell erklären.

Benveniste et. al. [23] bekräftigen das: „*Today, we see with some surprise that visual notations for synchronous languages have found their way to successful industrial use with the support of commercial vendors. This probably reveals that building a visual formalism on the top of a mathematically sound model gives actual strength to these formalisms and makes them attractive to users.*“

Es gibt jedoch auch eine Reihe offensichtlicher Nachteile visueller Sprachen. Insbesondere erschwert die Nicht-Linearität visueller Diagramme deren Erstellung und insbesondere Modifikationen. Um sich Überblick zu verschaffen, setzen Anwender visueller Sprachen oft die Papierform ein, wobei häufig schon das Drucken an sich ein Problem ist, und das Resultat den ganzen Schreibtisch, die Wand oder gar den Fußboden in Anspruch nimmt. Bei textuellen Sprachen reduziert sich dieser Aufwand hingegen auf das Durchblättern eines vergleichsweise kompakten Aktenordners, ein komplettes Auslegen eines Programms auf dem Fußboden stellt eher die Ausnahme dar. In einem Programmtext lassen sich leicht Elemente einfügen, verschieben oder löschen, ohne dass die Lesbarkeit des Dokuments insgesamt leidet. In einem visuellen Diagramm hingegen erfordert dass Einfügen oder Verschieben von Elementen oft aufwändige Editieroperationen, wenn das Diagramm weiterhin gut lesbar bleiben soll. Zitat eines Programmierers: „*I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it*“ [36].

Tatsächlich ist auch die inhärent „bessere Lesbarkeit“ graphischer Formalismen experimentell nur bedingt nachvollziehbar; eine Reihe von Studien weisen eher auf das Gegenteil hin [37, 36, 34]. Im Gegensatz zu einem künstlerischen Bild, welches ein Betrachter unvoreingenommen „auf sich wirken lässt“ um sich einen individuellen Gesamteindruck zu verschaffen, geht es bei visuellen Programmiersprachen darum, einen konkreten Sachverhalt präzise, effizient und für den Betrachter eindeutig zu vermitteln. Eine Frage ist also: drückt ein Bild für jeden Betrachter die gleichen Inhalte aus? Und wieviel Aufwand erfordert es, um diese Inhalte zu extrahieren? Geschriebener Text ist im wesentlichen Graphik mit einem sehr eingeschränktem Vokabular. Der geübte Leser führt hier jedoch einen Abstraktionsschritt durch, und nimmt Buchstaben nicht mehr als eigenständige graphische Elemente mit ihren individuellen Wahrnehmungseigenschaften wahr. Text wurde über Jahrhunderte als Medium zur Übermittlung technischer Informationen adaptieren; visuelle Sprachen sind hier noch vergleichsweise unterentwickelt.

Ein wesentliches Ergebnis kognitiver Untersuchungen ist, dass das Verständnis eines Diagramms nicht nur von den in der visuellen Sprache direkt definierten Sprachelementen abhängt, sondern auch sehr stark von *Sekundärnotationen*, wie dem Layout und weiteren typographischen Hinweisen. Erfahrene Leser nutzen typographische und andere Hinweise zum effizienten Textverständnis, ebenso wie Programmierer [35]; die gleichen Mechanismen greifen bei visuellen Sprachen.

Petre fasst das Verhalten von Anfängern und erfahrenen Entwicklern in einer Studie [37] wie folgt zusammen:

- Wenn äquivalente graphische und textuelle Repräsentationen nebeneinander zur Verfügung gestellt wurden, nutzten erfahrene Anwender fast immer die textuelle Darstellung als Anleitung zum Lesen der graphischen Darstellung.
- Anfänger hatten oft Mühe, in einem Diagramm zu erkennen, was wichtig ist – im Gegensatz zur verbreiteten Annahme, dass graphische Darstellungen diese Information offensichtlich machen.

Anfänger waren oft durch explizite Verbindungen verwirrt – wenn eine Linie existierte, wurde sie als relevant eingestuft.

- Experten nutzten beim Lesen von Diagrammen auch ihre Finger besser, wobei sie zum Teil alle zehn Finger einsetzten, um sich Punkte eines Schaltkreises zu „merken“.

Moher et al. [34] haben die Lesbarkeit unterschiedlicher Varianten von Petri-Netzen untersucht. Ergebnisse waren:

- In keinem Fall waren die graphischen Varianten besser lesbar als die textuellen äquivalente.
- Die Lesbarkeit der Diagramme hing stark vom Layout ab.

Trotzdem haben sich Programmierer, befragt über ihre Präferenzen bezüglich graphischer und textueller Sprachen, positiv über graphische Sprachen geäußert. Danach

- sind graphische Sprachen reichhaltiger, das heißt, mehr Information werden mit weniger Überflüssigem auf weniger Raum dargeboten;
- ergeben graphische Sprachen den „Gestalt“-Effekt und machen Strukturen sichtbarer;
- ist die Abstraktionsebene höher und näher an der Problemstellung;
- sind graphische Sprachen zugänglicher, leichter und schneller zu verstehen, leichter zu behalten;
- sind graphische Sprachen weniger formal und „nicht-symbolisch“;
- machen graphische Sprachen „mehr Spaß“.

Eine Feststellung ist, dass die *Illusion* der besseren Zugänglichkeit wichtiger sein könne als die Realität; schon der generell empfundene Reiz graphischer Sprachen als solcher kann ein wichtiger Motivator sein.

Petre [36] stellt fest, dass sich inhaltlich identische Entwürfe eines Anfängers und eines erfahrenen Anwenders darin unterscheiden, dass der Entwurf des Anfängers aufgrund des Layouts schwerer zu verstehen ist. Weiterhin wird festgestellt: „*It is time to recognize the impact of 'bad graphics'—of haphazard use of perceptual cues and secondary notations—mis-cueing, misleading, misreadig, and misunderstanding.*“ Dies ist insbesondere für den Entwurf sicherheitskritischer Systeme relevant, welche eine wesentliche und an Bedeutung zunehmende Anwendungsdomäne von Statecharts sind. Außerdem wird vermerkt: „*It appears that graphical notations can have a greater capacity to 'go wrong' than textual notations.*“

Für textuelle Sprachen gibt es eine Reihe wohldefinierter Richtlinien oder *Styleguides*, wie z. B. auch für das Schreiben von Programmen in C [33] oder Java [39]. Es gibt auch Beispiele solcher Richtlinien für graphische Sprachen, wie Statecharts [32]; diese sind aber im Vergleich zu den textuellen Styleguides generell recht unpräzise und geben wenig konkrete Hinweise zum Layout, sondern empfehlen z. B. Obergrenzen bezüglich der Zahl der Zustände pro Statechart. Ebenso gibt es für textuelle Sprachen Analyse- und Formatierungswerkzeuge (*Pretty Printer*), welche die Einhaltung von Entwurfsrichtlinien überprüfen bzw. erwirken; für graphische Programmiersprachen existiert dies erst sehr eingeschränkt.

Purchase et. al. [38] haben eine Studie durchgeführt, um den Zusammenhang zwischen der Erfüllung bestimmter ästhetischer Kriterien und der Lesbarkeit von UML Klassendiagrammen zu ermitteln. Mehrere Ergebnisse der Studie widersprachen den ursprünglichen Erwartungen. Purchase et. al. stellen fest, dass ein Layoutalgorithmus auch die Semantik der zugrundeliegenden Objekte beachten sollte.

Eine Reihe von Diagrammtypen nutzen auch die dritte Dimension zur komprimierten Darstellung großer Informationsmengen. Ware und Frank [40] haben kognitive Untersuchungen hinsichtlich der

hierdurch erreichbaren Erhöhung des Informationsgehalts durchgeführt. Unter Zuhilfenahme von Stereovision und interaktiver Animation (Möglichkeit zur Rotation der Struktur) ergab sich, dass eine Erhöhung um den Faktor drei realistisch ist; ohne diese Hilfsmittel war der Faktor niedriger, aber immer noch signifikant.

4.3 Beispielhafte ästhetische Kriterien für Statecharts

Eine beispielhafte Reihe ästhetischer Kriterien für Statecharts sind die folgenden.

- Für Zustände:
  - Zustände sollten nicht zu nah beieinander sein.
  - Zustände sollten nicht zu weit voneinander entfernt sein.
  - Zustände sollten auf einem Raster platziert werden.
  - Zustände sollten gleichverteilt sein.
- Für Transitionen:
  - Transitionen sollten nicht zu kurz sein.
  - Transitionen sollten nicht zu lang sein.
  - Die Länge von Transitionen sollte wenig variieren.
  - Transitionen sollten sich möglichst wenig überschneiden.
  - Transitionen sollten möglichst wenig Knicke oder Biegungen enthalten.
- Weiterhin:
  - Zustände und Transitionen sollten einen Mindestabstand haben
  - Geringe Fläche $\Longrightarrow$ homogene optische Dichte.
  - Das Verhältnis zwischen Höhe und Breite sollte ausgeglichen sein.
  - Symmetrische Informationen sollten sich in symmetrischem Layout spiegeln.
  - Semantisch zusammenhängende Strukturen sollten durch optische Nähe geclustert werden.

Literatur

[23] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[24] Therese C. Biedl and Michael Kaufman. Area-efficient static and incremental graph drawings. In *Proceedings of the 5th Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer Verlag, 1997.

ANNOTATION: The paper presents algorithms for two scenarios. In the static scenario, the graph is given completely in advance. In the incremental scenario, the graph is given one node at time, and the placement of previous node cannot be changed for later nodes. In each scenario, every edge gets at most one bend, thus the total number of bends is at most m edges.

[25] Franz J. Brandenburg, Michael Jünger, and Petra Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Research Report Number: MPI-I-97-1-007*, Informatik-Spektrum 20(4):199–207, 1997.

ANNOTATION: Through the use of some selected application examples, typical problems and solutions, the paper provides an introduction into drawing of graphs.

[26] Jürgen Branke. Dynamic Graph Drawing. In M. Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

ANNOTATION: The chapter tries to elucidate the concept of the mental map and surveys different approaches that authors have suggested to capture it. A number of frameworks and algorithms proposed in the literature for dynamic graph drawing are presented.

[27] Micheal K. Coleman and D. Stott Parker. Aesthetics-based Graph Layout for Human Consumption. *Software – Practice and Experience*, 26(12):1415–1438, December 1996.

[28] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, October 1996.

[29] D. Harel and G. Yashchin. An Algorithm for Blob Hierarchy Layout. *The Visual Computer*, 18:164–185, 2002.

ANNOTATION: The paper presents an algorithm for the aesthetic drawing of basic hierarchical blob structures (rounded-corner rectilinear shapes). Several criteria for aesthetics are formulated, the paper discusses their motivation, the methods of implementation and the algorithm's performance.

[30] Tobias Kloss. Flexibles und Automatisiertes Layout von Statecharts. Studienarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, July 2003.

[31] Corey Kosak, Joseph Marks, and Stuart Shieber. Automating the layout of network diagrams with specified visual organization. *Transactions on Systems, Man and Cybernetics*, 24(3):440–454, March 1994.

[32] Th. Kreppold. Modellierung mit Statemate MAGNUM und Rhapsody in Micro C. Berner & Mattner Systemtechnik GmbH, Otto-Hahn-Str. 34, 85521 Ottobrunn, Germany, Dok.-Nr.: BMS/QM/RL/STM, Version 1.4, August 2001.

[33] Christopher M. Lott. C and C++ style guides. `http://www.chris-lott.org/resources/cstyle/`.

[34] Thomas G. Moher, David C. Mak, Brad Blumenthal, and Laura M. Leventhal. Comparing the comprehensibility of textual and graphical programs: The case of petri nets. In *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161. Ablex Publishing, 1993.

[35] Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:295–341, 1987.

[36] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

ANNOTATION: The paper investigates how textual and visual representations for software differ in effectiveness. One finding is that the differences lie not so much in the textual-visual distinction as in the degree to which specific representations support the conventions experts expect.

[37] Marian Petre and Thomas R. G. Green. Learning to read graphics: Some evidence that "seeing" an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4(1):55–70, 1993.

[38] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In *ACM International Conference Proceeding Series archive, Australian symposium on Information visualisation*, pages 129–137, 2001.

ANNOTATION: This paper describes a study which aimed to identify the most important aesthetics for the layout of UML class diagrams from a human comprehensive point of view. The results suggest that for specific domains, the actual semantics of the given graph may need to be considered before an appropriate graph drawing can be produced.

[39] Sun Microsystems, Inc. Code conventions for the java programming language. `http://java.sun.com/docs/codeconv/`.

[40] Colin Ware and Glenn Franck. Viewing a graph in a virtual reality display is three times as good as a 2D diagram. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proc. IEEE Symp. Visual Languages, VL*, pages 182–183, 4–7 October 1994.

5 Layout-Verfahren

Das Layout von Statecharts kann als graphtheoretisches Problem formuliert werden. Das automatische Layout von Graphen ist ein inzwischen klassisches Gebiet der Informatik [48, 60], motiviert zum Beispiel durch VLSI-Layout oder die Darstellung von Datenstrukturen. Eine Vielzahl von Werkzeugen und Libraries ist hierzu bereits verfügbar [42, 54, 61].

5.1 Layout-Verfahren für statische Systeme

Harel und Yashchin [52] definieren einen Algorithmus für das Layout hierarchischer *blob structures*, welche aus verschachtelten, abgerundeten Rechtecken bestehen. Dies ist eine Generalisierung gegenüber den klassischen Annahmen des Graph-Layouts hinsichtlich der Knoten, welche nicht mehr punktförmig sein müssen, sondern den hierarchischen Zuständen von Statecharts entsprechen. Das Layout von Transitionen und Labels bleibt allerdings ausgeklammert.

Ein sehr einfacher, aber in der Praxis sehr effektiver Ansatz für das Layout von Statecharts wurde in dem Spezifikationswerkzeug RSML entwickelt [53, 57]: hier werden die Zustände einer Hierarchieebene jeweils horizontal bzw. vertikal angeordnet, dies mit den Hierarchieebenen alternierend. RSML beschränkt sich dabei jedoch auf die Zustände und Transitionen, Transitionslabels werden nicht platziert, sondern befinden sich in separaten, textuellen Beschreibungen, den *AND/OR Tables*.

Castelló et al. haben eine Umgebung für die Visualisierung von Statecharts entwickelt, welches Zustände in Schichten gruppiert und als Baumstruktur darstellt [47]. Zum automatischen Layout von Bäumen gibt es eine Reihe von Verfahren [48]. Es gibt bei Statecharts eine Hierarchie von Ober- und Unterzuständen, welche sich als reine Baumstruktur (ohne Vorwärts-, Rückwärts- oder Kreuzungskanten) darstellen lässt, wobei die Kanten nicht den Zustandstransitionen entsprechen, sondern den

Hierarchierelationen. Bei der Darstellung von Statecharts liegt das Hauptaugenmerk jedoch nicht auf der Visualisierung dieser Hierarchiebeziehung, sondern auf der Visualisierung der Zustände und Transitionen einer bestimmten Hierarchieebene. Abgesehen von gewissen Einschränkungen sind alle Arten von Transitionen möglich. Eine solche Einschränkung ist zum Beispiel, dass Transitionen nicht in Unterzustände hineinspringen dürfen. Zyklische Transitionen sind generell jedoch möglich, so dass hier keine eindeutige Baumstruktur existiert. Ein sich daraus ergebendes Teilproblem für das Layout von Statecharts ist, dass die Zuordnung der Zustände an Schichten generell nicht eindeutig ist. Ein mögliches Optimalitätskriterium ist die Minimierung der Anzahl von rückwärts gerichteten Transitionen, welches ein klassisches NP-hartes Problem ist [55, 1]. Gansner et al. setzen einen Simplex-Algorithmus für das *Layering* der Knoten ein und erzeugen Splines für die Transitionen [50, 51]; eines ihrer Anwendungsbeispiele ist ein endlicher Automat, jedoch ohne die Hierarchie und Parallelität von Statecharts. Zustandslabels werden als virtuelle Knoten betrachtet und neben den Transitionen platziert. Diese Arbeiten beschränken sich jedoch auf statische Sichten.

5.2 Layout-Verfahren für interaktive Systeme

Interaktive Anwendungen stellen besondere Anforderungen an das Verfahren, das zur Positionierung von Objekten verwendet wird. Das durch das Statechart repräsentierte Modell kann sich ändern, Teile des Statecharts können skaliert oder in verschiedenen Abstraktionsstufen dargestellt werden. Der einfachste Weg wäre, das Layout nach jeder Änderung vollständig neu zu berechnen. Dieser Ansatz ist jedoch äußerst ineffizient; außerdem besteht die Gefahr, dass das Statechart stark verändert wird.

Bei der Arbeit mit einem Statechart erstellt der Nutzer unbewusst eine kognitive Karte (*mental map*), die Informationen über die Struktur und die Bedeutung eines Statecharts enthält. Jede Änderung der Darstellung erfordert eine Anpassung der kognitiven Karte; die Änderungen sollten möglichst gering gehalten werden, um den Anpassungsprozess zu vereinfachen. Die Nutzung dynamischer *Graph-Drawing*-Algorithmen ermöglicht es, Änderungen eines Statecharts oder der zugrunde liegenden Darstellungskonventionen bei der Berechnung des Layouts zu berücksichtigen. Bei dynamischen *Charts* können solche Algorithmen für die Berechnung der Darstellung einzelner Simulationsschritte verwendet werden.

Es existieren zwei grundlegende Ansätze, die Anpassung der kognitiven Karte nach Statechart-Änderungen zu unterstützen:

1. Änderungen können hervorgehoben werden, damit der Nutzer sie schneller erkennt. Animationen, d. h. Überblendungen vom vorhergehenden zum aktuellen Statechart, haben sich in dieser Hinsicht besonders bewährt.

2. Der Aufwand zur Anpassung der kognitiven Karte lässt sich minimieren, indem die Änderungen so gering wie möglich gehalten werden. Dabei muss im Allgemeinen ein Kompromiss zwischen minimalen Änderungen und ästhetischen Kriterien eingegangen werden.

Die Eigenschaft eines Layout-Verfahrens, dass Änderungen der Statechart-Struktur möglichst geringe Änderungen des Layouts nach sich ziehen, wird *dynamische Stabilität* genannt. Der Bewahrung von Zustandspositionen kommt eine besondere Bedeutung zu, da sie in der kognitiven Karte als feste Orte gespeichert werden; Transitionen werden meist nur temporär betrachtet. Möglichkeiten zur Gewährleistung der dynamischen Stabilität sind die folgenden.

Beschränkung der Änderungen auf einen Teil des Layouts: Beim Einfügen neuer Zustände oder Transitionen kann beispielsweise gefordert werden, die Position der anderen Objekte nicht zu verändern. Auf diese Weise können die bestehenden Bereiche der kognitiven Karte bewahrt werden; allerdings ist es selten möglich, alle ästhetischen Kriterien und Layout-Konventionen zu berücksichtigen. Diesem Problem kann begegnet werden, indem Anpassungen nicht strikt verboten, sondern auf eine gewisse Umgebung der geänderten Objekte begrenzt werden.

Metriken zur Erfassung der Divergenz zweier Statecharts: Solche Metriken sind so gestaltet, dass die ermittelten Abweichungen den Aufwand zur Anpassung der kognitiven Karte möglichst realistisch widerspiegeln. Viele Verfahren basieren auf dem direkten Vergleich von Koordinaten; ein Beispiel ist die Berechnung der Summe der Abstände aller Zustände zwischen Ursprungs- und Ziel-Statechart. Rotationen, Verschiebungen oder Skalierungen von Statecharts oder Teil-Statecharts würden in diesem Fall jedoch als gravierende Änderungen bewertet werden. Es ist empfehlenswert, derartige Bearbeitungsschritte durch den Algorithmus speziell zu berücksichtigen.

Besonders wirkungsvoll können Kombinationen inkrementeller und statischer Layout-Algorithmen sein. Während der Ausführung von Editieroperationen wird z. B. das Statechart durch ein inkrementelles Layout-Verfahren angepasst. In die Berechnung wird dabei nur der durch die Operation geänderte Teil des Statecharts einbezogen. Nach Beendigung des Bearbeitungsschrittes wird ein vollständiges Layout des gesamten Statecharts durchgeführt. Die Änderungen des Statecharts werden besonders berücksichtigt, um die dynamische Stabilität zu erhalten.

5.3 Techniken zur Layoutspezifikation

Es können zwei grundlegende Ansätze unterschieden werden, um Anforderungen an die Positionierung von Statechart-Objekten zu beschreiben [17]:

- Der *algorithmische Ansatz* nutzt einen parametrisierten Optimierungsalgorithmus zur Berechnung der Positionen. Ästhetische Kriterien werden in Form von parametrisierten Optimierungszielen in den Algorithmus integriert.

  Vorteil: rechentechnische Effizienz.

  Nachteil: erneute Berechnung des gesamten Statecharts.

- Beim *deklarativen Ansatz* wird die Darstellung des Statecharts mittels einer vom Nutzer festgelegten Menge von Bedingungen (*Constraints*) spezifiziert. Die Positionen der Objekte können dann z. B. durch einen *Constraint*-Löser berechnet werden.

  Vorteil: ein *Constraint*-Löser ist für die Korrektur des Layouts verantwortlich.

  Nachteil: komplexe Ausdrücke für natürliche ästhetische Kriterien.

Literatur

[41] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[42] aiSee. Company homepage. `http://www.aisee.com/`.

[43] T. C. Biedl, B. P. Madden, and I. G. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In G. D. Battista, editor, *Graph Drawing (Proceedings GD '97)*, pages 391–402. Springer Verlag, 1997. Lecture Notes in Computer Science 1353.

ANNOTATION: This paper presents orthogonal graph drawings from a practical point of view. Orthogonal drawing algorithms are described that work for input graph of any degree. The three-phase method is a novel generic technique to create high-degree orthogonal drawings. This approach simplifies the description and implementation of orthogonal graph drawing, and can be applied to global as well as interactive and incremental settings.

[44] Therese C. Biedl and Michael Kaufman. Area-efficient static and incremental graph drawings. In *Proceedings of the 5th Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer Verlag, 1997.

ANNOTATION: The paper presents algorithms for two scenarios. In the static scenario, the graph is given completely in advance. In the incremental scenario, the graph is given one node at time, and the placement of previous node cannot be changed for later nodes. In each scenario, every edge gets at most one bend, thus the total number of bends is at most m edges.

[45] Jürgen Branke. Dynamic Graph Drawing. In M. Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

ANNOTATION: The chapter tries to elucidate the concept of the mental map and surveys different approaches that authors have suggested to capture it. A number of frameworks and algorithms proposed in the literature for dynamic graph drawing are presented.

[46] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. A Framework for the Static and Interactive Visualization for Statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.

ANNOTATION: This paper represents a framework for the automatic generation of layouts of Statecharts diagrams. The framework is based on several techniques that include hierarchical drawing, labeling and floor planning, designed to work in a co-operative environment. The resulting drawings enjoy several properties: hierarchical decomposition of states into substates; a low number of edge crossings; uniform aspect ratio; small area requirements. The paper also represents techniques for interactive operations and incremental drawing.

[47] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. ViSta. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing : 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 481–482. Springer, 2002.

[48] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994.

[49] Carsten Friedrich and Michael E. Houle. Graph Drawing in Motion II. In P. Mutzel et. al., editor, *GD 2001: Graph Drawing (LNCS 2265)*. Springer Verlag, 2001.

[50] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[51] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1234, 2000.

[52] D. Harel and G. Yashchin. An Algorithm for Blob Hierarchy Layout. *The Visual Computer*, 18:164–185, 2002.

ANNOTATION: The paper presents an algorithm for the aesthetic drawing of basic hierarchical blob structures (rounded-corner rectilinear shapes). Several criteria for aesthetics are formulated, the paper discusses their motivation, the methods of implementation and the algorithm's performance.

[53] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[54] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Springer, October 2003.

[55] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[56] Oliver Köth. Semantisches Zoomen in Diagrammeditoren am Beispiel von UML. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001.

ANNOTATION: This work deals with presentation and handling of large diagrams in graphical editors, which were produced by the DIAGEN system. It is shown that work with large graphical representations requires particular support by the editor to obtain lucidity and manageability.

[57] Kurt Partridge. Personal Communication, 2003.

[58] J. Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. D. Battista, editor, *Graph Drawing (Proceedings GD '97)*, pages 415–424. Springer Verlag, 1997. Lecture Notes in Computer Science 1353.

ANNOTATION: The paper presents an extension of the Sugiyama algorithm [59] together with orthogonal drawing. It includes an algorithm for incremental placement of classes.

[59] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[60] Ioannis G. Tollis, Giuseppe Di Battista, Peter Eades, and Roberto Tamassia. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[61] Thomas Willhalm. Software packages. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*, chapter A, pages 274–281. Springer-Verlag, Berlin, Germany, 2001.

6 Verfügbare Werkzeug-Infrastrukturen

Es existieren eine Reihe von Werkzeugen und Software-Bibliotheken, welche in diesem Themenfeld relevant sind [42, 54, 61]. Es folgt ein Abriss der naheliegendsten Systeme.

Für das Erstellen graphischer Editoren gibt es eine Reihe von Werkzeugen [54], zum Beispiel HotDraw [66, 65], ein in Smalltalk geschriebenes Framework für das Zeichnen strukturierter Graphen basierend auf Visual Works, Unidraw [79, 78], ein C++-Framework für das Erstellen graphischer Editoren, GenGEd [63, 64] welches auf algebraischen Graphtransformationen und Constraint Solving basiert, sowie VisPro [83], welches auf kontext-sensitiven *Reserved Graph Grammars* [82] basiert. Im Folgenden gehen wir auf zwei Werkzeuge gesondert ein, DIAGEN und ArgoUML.

6.1 DiaGen

Einen aufgrund seiner Flexibilität besonders vielversprechenden Ansatz verfolgt DiaGen [70, 69], welches auf Hypergraph Grammatiken basiert. Aus einer XML-Spezifikation heraus werden Diagrammeditoren generiert, welche Syntax-geleitetes Editieren erlauben. Die in Abschnitt 3.6 beschriebene sFK-Darstellung [17] basierte auf DiaGen, und auch ein Statechart-Editor mit einer einfachen Simulationsmöglichkeit wurde hiermit bereits umgesetzt [71]. DiaGen selbst basiert auf Java und ist frei verfügbar. Jeder mit DiaGen erstellter Editor besteht aus folgenden Modulen (s. Abb. 7):

Diagramm:

- Form und Position der Komponenten des Diagramms;
- enthält Hypergraphen-Modell des Diagramms.

Benutzerinterface:

- ermöglicht Interaktion mit dem Editor;
- Darstellung und Manipulation von Komponenten.

Analyse:

- erkennt Zusammenhänge zwischen den einzelnen Diagrammteilen;
- nach Analyse wird eine interne Repräsentation des Diagramms (Semantik) erzeugt.

Layout:

- passt Lage und Form der Diagrammkomponenten an;
- greift auf Diagrammanalyse zurück.

Diagrammtransformation:

- programmgesteuerte Modifikationen des Diagramms.

Das Problem, die Geometrie des Diagramms bei Fokus-und-Kontext-Darstellung anzupassen, ist relativ einfach zu lösen: Das Layoutmodul, das ohnehin dafür zuständig ist, sprachspezifische Layoutkorrekturen vorzunehmen, kann auch dazu genutzt werden, das Diagrammlayout anzupassen, wenn Komponenten durch Abstraktion oder Verfeinerung ihre Größe geändert haben. Eine nichtlineare Verzerrung der Darstellung findet damit nicht statt.

Abbildung 8 zeigt als Beispiel eine sFK-Darstellung eines UML-Beispieldiagramms mit dem Element Kunde als Fokus. Dieses Element ist in allen Details sichtbar, während die umliegenden Kontextelemente abstrahiert worden sind.

Zur Ausführung solcher Diagrammveränderungen bietet es sich an, das bereits vorhandene Transformationsmodul zu erweitern. Das dabei notwendige Entfernen und Einfügen von Diagrammkomponenten durch Transformationen ist als Grundbestandteil jeder Diagrammänderung bereits implementiert. Da das Diagramm selbst und nicht nur seine Darstellung verändert wird, ist eine abstrahierende Zoom-Transformation mit einem Informationsverlust verbunden. Sie muss natürlich trotzdem umkehrbar sein; es ist daher notwendig, zusätzliche Sprachmittel für die Definition von Transformationen einzuführen, um bei der Abstraktion ausgeblendete Information zu speichern und später wieder zu rekonstruieren. Durch die Nutzung sprachspezifischer Diagrammtransformationen zur Abstraktion, welche von der Programmiererin definiert werden, ist auch die Berücksichtigung spezifischer Gegebenheiten der Diagrammsprache auf einfache Weise möglich. [17]

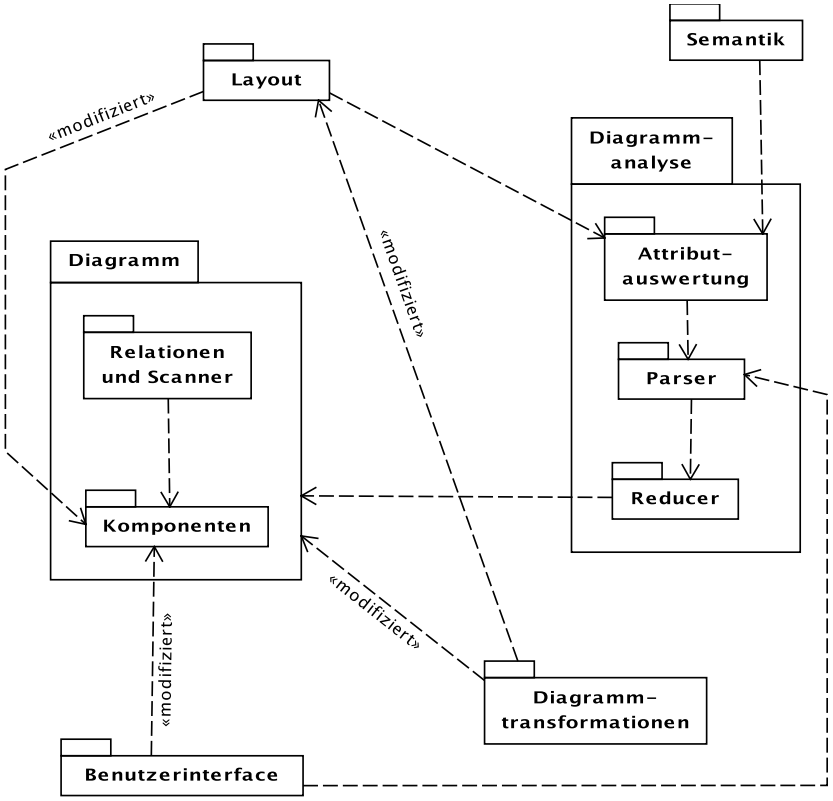


Abbildung 7: Interne Struktur eines DiaGen-Editors.

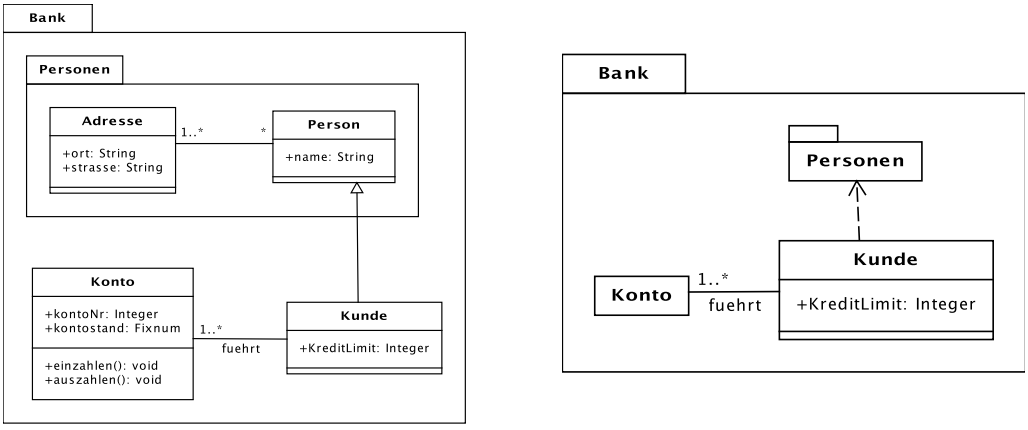


Abbildung 8: Sematische Fokus-und-Kontext-Darstellung eines Diagramms nach Koeth [17].

6.2 ArgoUML

Es existieren bereits eine Vielzahl von Modellierungswerkzeugen, welche graphische Editoren beinhalten, aber im Gegensatz zu den oben beschriebenen Werkzeugen weniger auf das Zeichnen sondern auf die eigentliche Systemmodellierung abzielen. Modellierungswerkzeuge müssen also für die graphische Modellierungssprache eine bestimmte Semantik implementieren, und bieten generell die Möglichkeit zur Simulation und Codesynthese. Als ein Beispiel sei hier ArgoUML [10] genannt, ein UML-Werkzeug mit einem besonderen Schwerpunkt auf dem Modellierungsprozess an sich und dessen kognitive Aspekte. ArgoUML verwendet integrierte Agenten (*design critics*), welche den Entwurf kontinuierlich überprüfen und bewerten; dies ist motiviert durch Schoens Theorie der *reflection-in/on-action* [77], nach welcher Modellierer Synthese und Analyse in einem Entwurfsprozess kontinuierlich durchmischen. Prädikate (*criticism control mechanisms*) steuern den Einsatz der *design critics*. ArgoUML ergänzt die durch UML vorgegebenen Diagrammtypen um aufgabenspezifische Sichten (*task-specific views*). Tabellensichten (*opportunistic table views*) geben aufgabenspezifische, komprimierte Sichten von Entwurfselementen; zum Beispiel kann ein Statechart als eine Tabelle von Transitionen mit Quelle, Senke, Triggern etc. dargestellt werden. Als Erweiterung der klassischen, starren Baumstrukturen, welche von CASE-Werkzeugen zur Orientierung angeboten werden, bietet ArgoUML eine Auswahl an Baumdarstellungen (*navigational perspectives*). So können zum Beispiel Klassenstrukturen einschließlich der in ihnen enthaltenen Zustände angezeigt werden, alternativ kann dies auch zusammen mit den enthaltenen Transitionen geschehen; darüberhinaus können zum Beispiel auch Baumstrukturen mit Zuständen und Nachfolgezuständen generiert werden. Eine Werkzeugkomponente (das *broom alignment tool*) erlaubt das Ausrichten graphischer Elemente, zur Unterstützung des Layouts und damit verbundener Sekundärnotationen; eine Anwenderstudie [75] ergab, dass der Einsatz dieses Werkzeugs die Anzahl der für die graphische Operationen erforderlichen Mausbewegungen signifikant verringerte. Designs werden in XMI abgespeichert [74]; jedoch werden graphische Informationen nicht mit den von XMI vorgegebenen Tags codiert, sondern in PGML (*Precision Graphics Markup Language* [73]). Das *Graph Editing Framework* (GEF) ist eine eigenständige Komponente zum Editieren verbundener Graphen. *Graph Models* bilden graphische Objekte auf Datenstrukturen ab, analog zu den Mediatoren des Java Swing Toolkits [80].

Literatur

[62] aiSee. Company homepage. `http://www.aisee.com/`.

[63] Roswitha Bardohl. GenGEd – A visual environment for visual languages. *Science of Computer Programming, Special Issue of GraTra '00*, 2002.

[64] GenGED. Project homepage. `http://tfs.cs.tu-berlin.de/~genged/`.

[65] HotDraw. Project homepage. `http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html`.

[66] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[67] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Springer, October 2003.

[68] Oliver Köth. Semantisches Zoomen in Diagrammeditoren am Beispiel von UML. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001.

ANNOTATION: This work deals with presentation and handling of large diagrams in graphical editors, which were produced by the DIAGEN system. It is shown that

work with large graphical representations requires particular support by the editor to obtain lucidity and manageability.

[69] M. Minas. XML-based specification of diagram editors. In R. Bardohl and H. Ehrig, editors, *Uniform Approaches to Graphical Process Specification Techniques (UNIGRA '03), Satellite Workshop to ETAPS 2003*, volume 82, Warsaw, Poland, April 2003. Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers.

ANNOTATION: This paper describes a graphical specification tool for DIAGEN, a diagram editor generator based on hypergraph grammars and hypergraph transformation. The specification tool simplifies the process of specifying and generating diagram editors. It uses an XML-based specification language, and it is an extension of a generic XML editor which offers syntax-directed editing based on the DTD, i.e., syntax specification, of the underlying XML-based language.

[70] M. Minas and J. Gottschall. Specifying animated diagram languages. In *International Workshop on Theory of Visual Languages (TVL '97)*, Capri, Italy, September 1997.

ANNOTATION: The paper discusses dynamic diagram languages with animators, i. e. animated dynamic diagram languages, and how to specify them. Specifications of diagrams are based on an internal hypergraph model and their syntax on hypergraph grammars. Animation of diagrams are described as sequences of hypergraph transformations which control the animation of graphical objects and which are specified using hypergraph rewriting rules. The paper discusses several concepts for the animation of graphical objects. These concepts have been realized as an animation framework. Specification and framework are extensions of DIAGEN, an existing framework and generator providing diagram editors for mainly static diagram classes.

[71] Mark Minas. Specifying statecharts with diagen. HCC '01 – 2001 IEEE Symposia on Human-Centric Computing Languages and Environments, Symposium on Visual Languages and Formal Methods, Statechart Modeling Contest, September 2001.

[72] Mark Minas and Berthold Hoffmann. Specifying and Implementing Visual Process Modeling Languages with DiaGen. In *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

ANNOTATION: This paper describes how a diagram language can be specified based on graphs, graph grammars, and transformation rules, and how the diagram editor generator DIAGEN generates a diagram editor from such a specification. DIAGEN can be applied to practically every visual language, and to visual process modeling languages in particular. This is demonstrated with an editor and animator for statecharts.

[73] Precision graphics markup language (pgml). World Wide Web Consortium Note 10-April-1998.

[74] Object Management Group. Xml metadata interchange (xmi) specification, version 2.0, May 2003. `http://www.omg.org/docs/formal/03-05-02.pdf`.

[75] J. Robbins, M. Kantor, and D. Redmiles. Sweeping away disorder with the broom alignment tool. In *Proceedings on Human Factors in Computing Systems (CHI '99)*, May 1999.

[76] Jason Robbins and David Redmiles. Cognitive support, uml adherence, and xmi interchange in argo/uml. *Journal of Information and Software Technology. Special issue: The Best of COSET '99*, 42(2):79–89, 2000.

[77] Donald A. Schön. *The Reflective Practitioner. How professionals think in action.* Basic Books, London, 1983.

ANNOTATION: An influential book that examines professional knowledge, professional contexts and reflection-in-action. The author examines the move from technical rationality to reflection-in-action and describes the process involved in various instances of professional judgement.

[78] Unidraw. Project homepage. `http://www.ivtools.org/ivtools/unidrawinfo.html`.

[79] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transaction on Information Systems*, 8(3):237, 1990.

[80] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* Addison-Wesley, July 1999.

[81] Thomas Willhalm. Software packages. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*, chapter A, pages 274–281. Springer-Verlag, Berlin, Germany, 2001.

[82] Da-Qian Zhang and Kang Zhang. Reserved graph grammar: A specification tool for diagrammatic VPLs. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pages 284–91, Isle of Capri, Italy, 23–26 1997.

[83] Ke-Bing Zhang, Mehmet A. Orgun, and Kang Zhang. Visual language semantics specification in the vispro system. In Jesse S. Jin, Peter Eades, David Dagan Feng, and Hong Yan, editors, *Pan-Sydney Workshop on Visual Information Processing (VIP2002)*, volume 22 of *Conferences in Research and Practice in Information Technology*, page 121, Adelaide, Australia, 2003. ACS.