Fachhochschule Dortmund
University of Applied Sciences and Arts
FB Informations- und Elektrotechnik

EPOS
Embedded core & Power Systems
GmbH & Co. KG

Masterthesis

Erweiterung eines Clocktree-Analyse-Tools zur Feststellung der strukturellen Äquivalenz von Clocktrees

Johannes Düperthal

Matrikelnummer 7085556

20. Februar 2019

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis

Zweitprüfer: Dr.-Ing. Stephan Brabender

Danksagung

Hiermit möchte ich mich bei allen Beteiligten des Projektes "Clocktree-Viewer" bedanken, die mich bei Problemstellungen stets unterstützt und mit ihrem Know-How meinen fachlichen Horizont erweitert haben. Ein ganz besonderer Dank gilt meinem Mentor Herrn Joachim Milde, der mich über die Grenzen dieser Arbeit hinaus inspiriert hat.

Zudem danke ich meinem Betreuer an der Fachhochschule Dortmund Prof. Dr. Michael Karagounis, der durch seine Begeisterung an der Mikroelektronik mein Interesse an diesem Themengebiet geweckt und gefestigt hat.

Erklärung

Hiermit versichere ich an Eides statt, dass die von mir vorgelegte Prüfungsleistung selbständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

Dortmund, 20.2.2019 _____

Unterschrift

Abstract

This master thesis describes the development of a component for clocktree comparison, which is implemented as an extension to an existing clocktree analysis tool. The existing program consists of a Tcl and a Qt application. All algorithms and the database, which contains the clocktree data, are part of the Tcl application. The user interface is realized by the Qt application, which is supplemented by a component which compares the clocktrees. The algorithm for this equivalence check is based on graph-theory. The clocktrees are transformed into tree-graphs to make the resulting structures comparable. The determined elements, which cause the difference, are colored in the Schematic-Viewer-Widget, which is already implemented in the existing application.

In dieser Masterthesis wird die Entwicklung einer Komponente zum Vergleich von Clocktrees beschrieben, die als Erweiterung in ein bestehendes Clocktree-Analyse-Tool integriert wird. Das bestehende Programm ist aus einer Tcl-Anwendung und einer Qt-Applikation aufgebaut. Alle Algorithmen und eine Datenbasis, welche die Daten zu den Clocktrees enthält, sind Teil der Tcl-Anwendung. Die Benutzeroberfläche wird durch eine Qt-Applikation realisiert, welche durch die Komponente für den Vergleich der Clocktrees ergänzt wird. Der Algorithmus für diesen Äquivalenzcheck basiert auf der Graphentheorie. Dazu werden die Clocktrees in Baum-Graphen transformiert, um die daraus resultierenden Strukturen vergleichbar zu machen. Die ermittelten Elemente, welche den Unterschied verursachen, werden in der Qt-Applikation, in einem Schematic-Viewer-Widget koloriert, das bereits in der bestehenden Applikation implementiert ist.

Inhaltsverzeichnis

Abkürzungen

API	Application Programming Interface
CG	Clockgate
CTA	Clocktree-Analyser
CTS	Clocktree-Synthese
DC	Design Compiler
EDA	Electronic-Design-Automation
FF	Flipflop
GUI	Graphical-User-Interface
ICC	IC Compiler
ID	Identifier
OID	Object Identifier
SoC	System-on-a-Chip
TCP/IP	Transmission Control Protocol/Internet Protocol

1 Einleitung

Moderne Mikroelektronik-Schaltungen werden mit Strukturgrößen von einigen Nanometern und in millionenfacher Anzahl auf Siliziumscheiben implementiert. Solche Systeme werden als System-on-a-Chip (SoC) bezeichnet. Um die Entwicklung solch komplexer Systeme zu ermöglichen, wird das System zuerst sehr abstrakt beschrieben und in einem bestimmten Ablauf von Schritten konkretisiert, sodass am Ende dieses Vorgangs die Abmessungen und physikalischen Eigenschaften der gesamten Chipstruktur vorliegen. Die Abfolge dieser Schritte wird als *Designflow* bezeichnet. Sie beginnt mit der Spezifikation von Chipanforderungen, durchläuft die Beschreibung der Schaltung in einer Hardwarebeschreibungssprache und wird in einer *Gatternetzliste* abgebildet. Daraufhin wird die Schaltung in ihren geometrischen Ausmaßen dargestellt und schließlich gefertigt. Für diesen Vorgang stehen Programme zur Verfügung, welche die Entwicklung bestmöglich automatisieren. Der Vorgang wird als Electronic-Design-Automation (EDA) und die Programme als EDA-Tools bezeichnet. Während der Konkretisierung der Schaltung orientieren sich die Tools an Vorgaben, die als *Constraints* bezeichnet und durch den Entwickler sowie der Spezifikation vorgegeben werden. Diese werden während des Entwicklungsvorgangs immer weiter verfeinert und konkretisiert. Im Rahmen des Projektes "Clocktree-Viewer" der Firma *EPOS embedded core & power systems GmbH & Co. KG*[1] -nachfolgend "EPOS" genannt- wird ein Tool entwickelt, das dabei helfen soll, solche *Constraints* zu definieren. Dabei wird ein bestimmter Teil der Schaltung analysiert, der sogenannte *Clocktree*. Dieser ist dazu da, um bestimmte Schaltungsteile synchron (sequentiell) zu betreiben. Das geschieht, indem ein Takt (Clock) an mehrere Bauteile der Schaltung angelegt wird. Bei jedem Impuls dieses Taktes werden alle verbundenen Bauteile nahezu zeitgleich ausgelöst. Weitere Informationen zur Entwicklung integrierter Schaltungen sind unter [7] zu finden.

Damit es möglich ist, den *Clocktree* zu analysieren und *Constraints* aus dieser Analyse abzuleiten, wird er zuerst mithilfe spezieller EDA-Tools aus der Schaltung extrahiert. Dazu werden bei *EPOS* die Programme IC Compiler (ICC)™, ICC™II oder Design Compiler (DC)® der Firma *SYNOPSYS*® verwendet. Abbildung 1.1 zeigt auf der linken Seite die Extraktion aus einer Schaltung. Im oberen Bereich ist der Ausschnitt einer digitalen Verschaltung eines SoCs zu sehen. Durch die Ansteuerung der Flipflops (FFs) durch einen Takt (CLK) kann diese Schaltung synchron zu einer anderen Schaltung, die von dem gleichen Takt angesteuert wird, betrieben werden. Im unteren Bereich

[1] Eine Tochterfirma der Infineon Technologies AG®

ist der extrahierte *Clocktree* aus der oberen Schaltung zu sehen. Nach der Extraktion bleiben nur noch Bauteile erhalten, welche direkt mit der *Clock* verbunden sind. Anschließend kann die Verschaltung analysiert werden. Auf der rechten Seite der Abbildung ist der komplette *Designflow* und auch die Stelle im *Flow* dargestellt, an der die Clocktree-Analyse stattfinden soll.

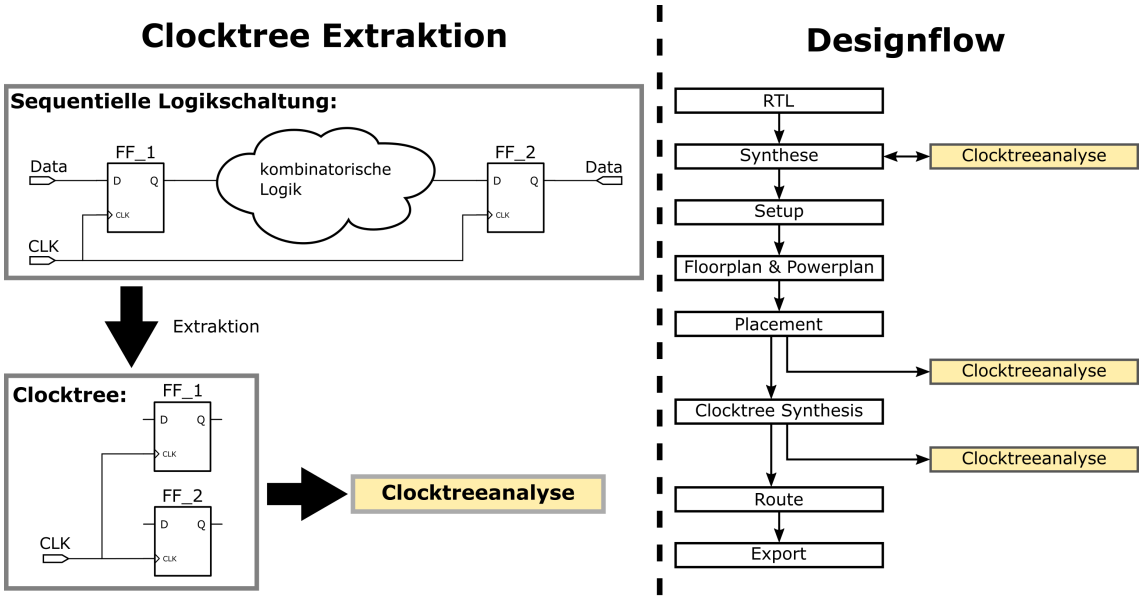


Abbildung 1.1: Clocktree-Extraktion und CT-Analyse im Designflow

Im Rahmen dieser Masterthesis wird ein bestehendes Tool, das in dem Projekt "Clocktree-Viewer" entwickelt wird, um eine zusätzliche Komponente erweitert. Mithilfe dieser Erweiterung soll es möglich sein, zwei Clocktrees miteinander zu vergleichen und die strukturellen Unterschiede herauszufinden. Die bestehende Software wird als Clocktree-Analyser (CTA) bezeichnet und besteht aus einer *Qt*-Applikation und einer *Tcl*-Anwendung, die eine Datenbasis beinhaltet, auf der das Programm aufbaut. Die darin enthaltenen Daten werden durch das Einlesen spezieller Dateien eingepflegt. Anhand dieser Daten kann eine Analyse des Clocktrees stattfinden. Die *Qt*-Applikation basiert auf der Schematic-Viewer-Engine *NLView™* der Firma *Concept Engineering*, mit der die Clocktrees dargestellt werden können. Abbildung 1.2 zeigt den grundsätzlichen Aufbau des Programmes. In dem Fenster "Qt-Anwendung" ist auch das Schematic-Widget mit einem *Clocktree* angedeutet. Zur Implementierung der Komponente, die in dieser Arbeit entwickelt wird, ist das Verständnis über den Softwareaufbau des bestehenden Programmes und der Umgang mit der Schematic-Viewer-Engine, erforderlich. Daher ist die Einarbeitung in diese Themengebiete Teil der Aufgabenstellung.

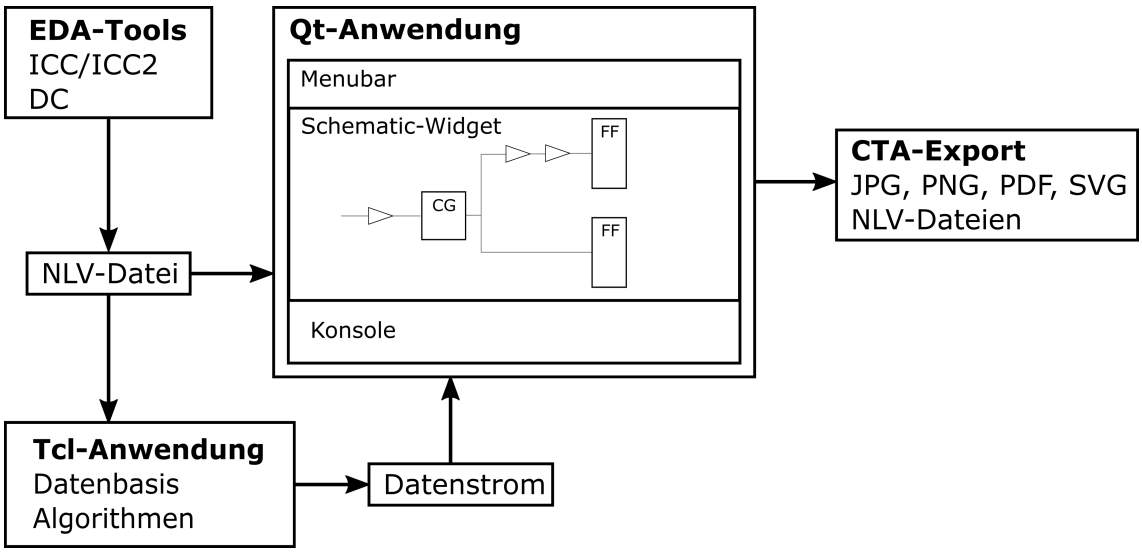


Abbildung 1.2: CTA Softwareaufbau

Die Komponente, zum Clocktree-Vergleich wird als *Comparetool* bezeichnet. Sie soll aus einer Benutzeroberfläche bestehen und auf einem Algorithmus basieren, mit dem es möglich ist, alle Elemente zu finden, die einen Unterschied in den beiden zu vergleichenden Clocktrees verursachen. Der Algorithmus soll auf der Graphentheorie basieren, mit der es unter anderem möglich ist, die Äquivalenz zwischen zwei Graphen festzustellen. Aus diesem Grund wird in diesem Dokument eine Einführung in das Themengebiet der Graphen gegeben. Um die Theorie auf Clocktrees anwenden zu können, müssen diese erst in Graphen umgewandelt werden. Dazu wird beim Einlesen der NLV-Dateien die Struktur der Clocktrees ermittelt und in die bestehende Datenbasis eingepflegt. Die Auswahl der Clocktrees soll in der Benutzeroberfläche durch zwei Auswahllisten stattfinden, in denen alle eingelesenen Clocktrees aufgeführt sind. Die Visualisierung beinhaltet die Ausgabe der Instanz-Bezeichnungen in einer gesonderten Konsole sowie die Kolorierung der Instanzen im Schematic-Widget.

Die zu vergleichenden Clocktrees stammen aus zwei unterschiedlichen Entwicklungsständen eines SoC-Designs. Die mithilfe des *Comparetools* ermittelten Unterschiede sollen dazu beitragen, die *Constraints* besser konkretisieren zu können. Die Differenzen entstehen durch Anpassungen der Schaltung, verursacht durch die Prozessierung des Entwurfes durch EDA-Tools. Solche Änderungen können zu Verstößen, wie beispielsweise "Timing violations", führen. Mithilfe der Analyse des *Comparetools* soll festgestellt werden, wo sich diese Änderungen befinden um anschließend im *Constraining*, spezifisch auf die betroffenen Elemente eingehen zu können.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zum Thema Clocktree erläutert. Dabei wird auf die Struktur des Clocktrees und dessen Bedeutung im Designflow sowie auf diesbezügliche Begriffe eingegangen. Weitere Informationen dazu sind unter [11] und [1] zu finden. Außerdem wird die im Tool verwendete Schematic-Viewer-Engine *NLView* vorgestellt und die wichtigsten Kommandos erläutert. Darüber hinaus erfolgt in diesem Kapitel die Einführung in das Thema Graphentheorie. Zur Verwendung von Graphen in Tcl existiert die Bibliothek "struct::tree". Auch diese wird in diesem Kapitel behandelt.

2.1 Clocktrees in integrierten Schaltungen

Der digitale Bestandteil von SoCs teilt sich in sequentielle und kombinatorische Logik auf. Als sequentielle Logik werden logische Schaltungen bezeichnet, welche das Speichern von Informationen beinhalten. Charakteristische Elemente dafür sind Flipflops. Kombinatorische Logik hingegen ist zustandslos und der Ausgang der Schaltung ist allein von den Eingängen abhängig. Abbildung 2.1 zeigt anhand von zwei Schaltungen, wie sich die beiden Logiktypen unterscheiden. Dem oberen Teil der Grafik ist auch zu entnehmen, dass die Flipflops mit dem Taktsignal Clock verbunden sind. Die Speicherzustände der Flipflops werden synchron zur Clock aktualisiert.

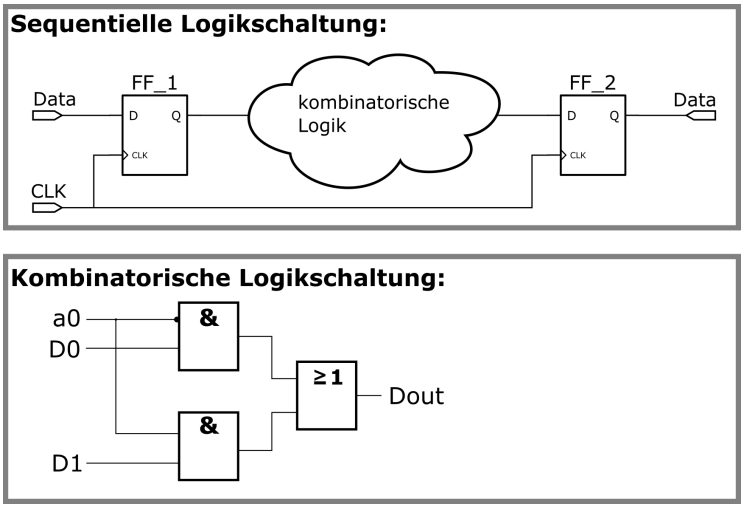


Abbildung 2.1: Sequentielle und kombinatorische Logik

Wenn alle Bauteile, welche mit der Clock verbunden sind, extrahiert werden, entsteht eine Schaltung die als Clocktree bezeichnet wird. Dazu gehören grundsätzlich Netze, Flipflops und eine Clocksource. Es können auch weitere Elemente vorkommen, die im weiteren Verlauf des Abschnitts erläutert werden. Ein solch grundlegender Aufbau ist auch in Abbildung 1.1 zu sehen. Die Clock wird durch einen Taktgeber, der außerhalb des SoCs betrieben wird, versorgt. Der Takt wird der Schaltung dann über ein sogenanntes *Pad* zugeführt. Dies ist ein analoges Element mit digitalem Interface. In diesem Fall wird die Clocksource daher als *Analog-Macro* dargestellt. Allgemein sind Macros wiederverwendbare Logik- oder Analogblöcke, die in einem SoCs integriert werden können. Die Elemente digitaler Schaltungen werden als Zellen bezeichnet. Diese Zellen können, wiederum in hierarchischen Zellen oder Hierarchien, zusammengefasst werden. Bei Betrachtung einer einzelnen Hierarchie entsprechen die Pins dieser Zelle den Clocksources und werden als *Ports* in das Schematic eingepflegt. In Abbildung 2.2 sind die beiden Clocksource-Typen, wie sie in *NLView* dargestellt werden, abgebildet. Außerdem ist eine hierarchische Anordnung von Zellen dargestellt. Es ist ebenfalls ersichtlich, dass hierarchische Zellen auch andere Hierarchien enthalten können.

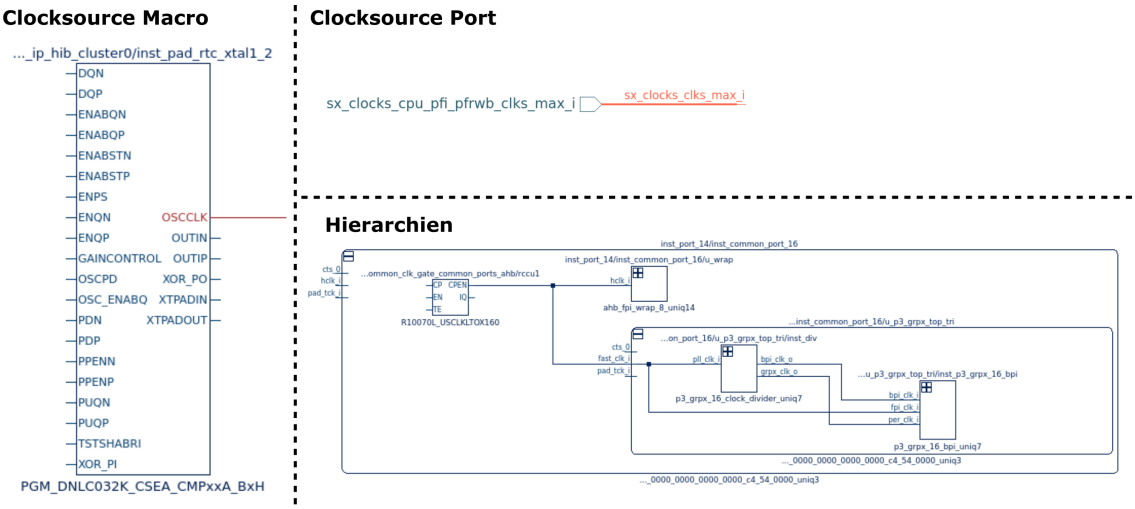


Abbildung 2.2: Clocksource-Darstellung und hierarchischer Aufbau digitaler Schaltungen

Eine wichtige Eigenschaft des Clocktrees ist die dynamische Leistungsaufnahme (*Dynamic Power*). Sie beschreibt die Leistung, die durch das Umladen der Transistor-Kapazitäten entsteht. Sie steigt mit der Anzahl der verwendeten Transistoren und der Frequenz des Signalwechsels. Durch die Abschaltung des Clock-Taktes an Logikgruppen, die nicht dauerhaft in Betrieb sind, kann diese Leistungsaufnahme gesenkt werden. Dazu werden sogenannte Clockgates (CGs) im Clocktree eingebaut, welche die Verbreitung der Clocks an solche Gruppen zu bestimmten Zeiten verhindern. In der Abbildung 2.3 sind ein Clockgate und vier abschaltbare Flipflops dargestellt. Das Clockgate verfügt über einen *Enable* (EN) Eingang. Über diesen wird die Zu- und Abschaltung des Taktes gesteuert.

Abbildung 2.3: Clockgate und Flipflops

Die Anzahl der Flipflops, in der oberen Grafik, ist vergleichsweise gering. In realistischen Entwürfen können mehrere hundert Flipflops hinter einem Clockgate geschaltet sein. Diese Clockgate/Flipflop-Gruppen können ebenfalls in großer Zahl mit dem gleichen Taktsignal verbunden sein. Diese dadurch entstehende Struktur eines Clocktrees ist in einer Schematic-Darstellung sehr unübersichtlich.

Aufgrund von Signalverzögerungen, die durch parasitäre Kapazitäten der Verbindungsleitungen verursacht werden, entsteht eine Asynchronität zwischen den Clock-Flanken, welche sich an unterschiedlichen Flipflops einstellen. Diese Verzögerung des Clock-Signals wird als *Skew* bezeichnet und ist im oberen Teil der Grafik 2.4 abgebildet. Um die Synchronität wieder herzustellen werden im schnelleren Pfad Buffer eingesetzt, sodass die Verzögerung in beiden Pfaden gleich ist. Dies ist im unteren Teil der Abbildung verdeutlicht. Die eingefügten Buffer sind ebenfalls Teil des Clocktrees. Im Designflow wird dieser Vorgang als Clocktree-Synthese (CTS) bezeichnet. Da auch in diesem Fall große Mengen dieser Elemente implementiert werden, wird das Schematic des Clocktrees durch die CTS noch unübersichtlicher.

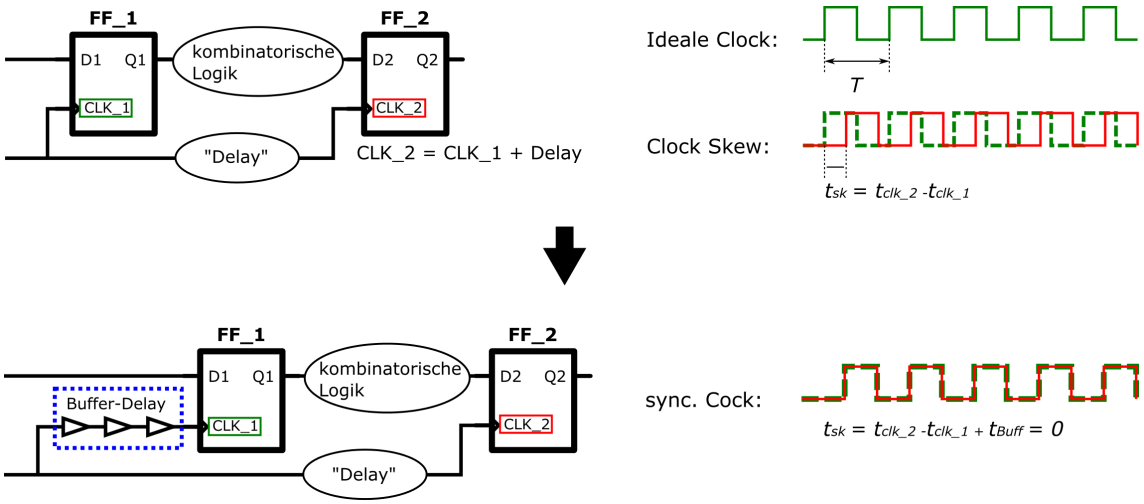


Abbildung 2.4: Skew und Clocktree-Synchronisation

Eine weitere wichtige Funktion des Clocktrees betrifft die Umschaltung zwischen unterschiedlichen Chip-Modi. Der Standardmodus wird als *Mission-Mode* bezeichnet. Ein weiterer Modus ist beispielsweise der *Test-Mode*. Bei diesem Modus wird die Funktionstüchtigkeit des Chips nach der Fabrikation festgestellt. Dazu werden die Flipflops als Schieberegister verschaltet, sodass Bitmuster durch diese Flipflop-Ketten geschoben werden können, die an einem bestimmten Eingang angelegt und an einem Ausgang ausgelesen werden können. So wird geprüft, ob die Flipflops richtig funktionieren. Während des Tests werden die FFs mit einer niedrigeren Frequenz als im funktionalen Modus betrieben. Um zwischen den verschiedenen Clock-Modi wechseln zu können, werden Multiplexer verwendet. Abbildung 2.5 zeigt einen solchen Multiplexer, der zwischen den beiden farbig gekennzeichneten Clocks, die an den Pins A0 und A1 anliegen, umschalten kann. Da die verwendeten EDA-Tools nicht wissen, wann welche Clock durchgeschaltet wird, werden in den Constraints sogenannte *Exception-Pins* gesetzt. Mit diesen kann eine Clock gestoppt werden. Das bedeutet, dass das Verhalten des Multiplexers in den Constraints so beschrieben ist, dass der Multiplexer eine andere Clock durchschaltet und nicht die Clock, die am *Exception-Pin* anliegt. Der grün markierte Pin A1 in Abbildung 2.5 entspricht einem derartigen *Exception-Pin*.

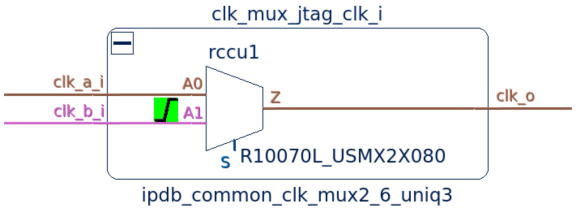


Abbildung 2.5: Multiplexer im Clocktree

In der CTS spielen die *Exception-Pins* eine besondere Rolle, da über die Schaltfunktion der Multi-

plexer die Pfade, welche die Clock über die Netze nimmt, verändert werden. Wenn die *Exception-Pins* falsch gesetzt sind oder an einigen Multiplexern nicht existieren, so kann es passieren, dass das CTS-Tool versucht, Pfade zu synchronisieren, die im regulären Betrieb nicht parallel betrieben werden. Dadurch werden überflüssige Buffer im Clocktree eingebaut, was den Anstieg der dynamischen Leistungsaufnahme des Chips zur Folge hat. Im Rahmen des Projekts bei *EPOS* soll diesem Effekt entgegengewirkt werden, indem das Setzen der *Exception-Pins* vereinfacht und besser visualisiert wird. Wenn ein *Exception-Pin* an einem Multiplexer fehlt, überlagern sich die Clocks am Ausgangs-Pin. Ein anderer Fehlerfall besteht darin, dass an zwei Eingängen des Multiplexers die gleiche Clock anliegt. Solche Ereignisse werden als *Rekonvergenz* bezeichnet. Das SYNOPSYS Tool *ICC*™ bietet die Möglichkeit, Reports zu exportieren, welche die vom Tool gesetzten *Exception-Pins* beinhalten.

2.2 Graphentheorie

Die Graphentheorie ist ein Teilgebiet der diskreten Mathematik, das sich mit abstrakten Strukturen und deren Zusammenhänge beschäftigt. Diese Strukturen werden als *Graphen* bezeichnet. Die Graphentheorie ist auf den Mathematiker Leonhard Euler und das Jahr 1736 zurückzuführen. Euler suchte eine Lösung für das Königsberger Brückenproblem, bei dem es darum ging, jede Brücke in Königsberg genau einmal zu überqueren und dabei einen Rundgang durch die Stadt zu machen. Abbildung 2.6 zeigt die Stadt mit den zu überquerenden Brücken. Mithilfe der Graphentheorie konnte Euler beweisen, dass es nicht möglich ist, die Brücken in einem Rundgang zu überqueren (v.g.l [2]).

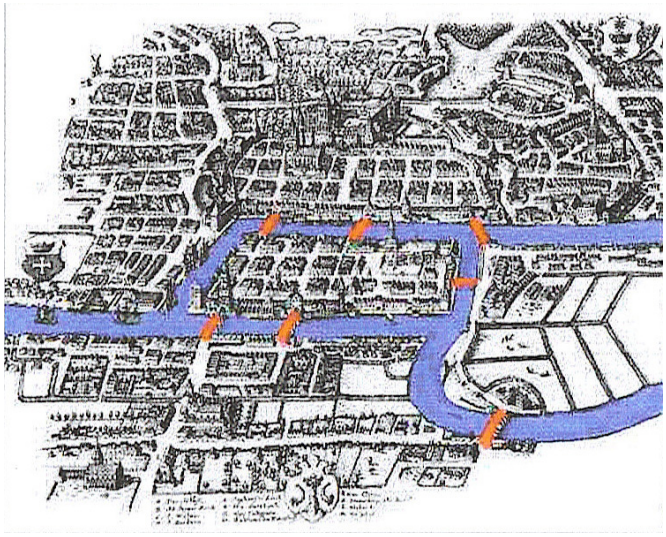


Abbildung 2.6: Königsberger Brückenproblem[6]

Neben diesem ersten Beispiel aus Königsberg gibt es zahlreiche weitere Anwendungsgebiete für

die Graphentheorie, unter anderem in der Routenplanung, der Informatik und im Projektmanagement. Auch für die Darstellung elektrischer Schaltungen und Netzwerke wird die Graphentheorie verwendet. Die Theorie findet meistens in Form von Algorithmen Anwendung, bei denen es beispielsweise darum geht, die kürzeste Verbindung zwischen zwei Objekten zu finden. Die Angaben in diesem Kapitel über die Graphentheorie sind [8] entnommen und sinngemäß in dieses Dokument überführt.

2.2.1 Grundbegriffe

In der Graphentheorie treten einige neue Definitionen auf, die in diesem Abschnitt erläutert werden.

Graph, Knoten, Kante, Weg:
Ein **Graph** besteht aus einer nichtleeren Menge von *Knoten*. Diese Knoten sind durch sogenannte *Kanten* miteinander Verbunden. Eine Verbindung zwischen zwei Knoten, die mehrere Knoten und Kanten beinhaltet wird als Weg oder Pfad bezeichnet. In der Grafik 2.7 sind einige Graphen abgebildet.

Digraph, Pfeile:
Ist den Kanten eines Graphen eine Richtung zugeordnet, so werden diese als *Pfeile* bezeichnet. Der Graph ist dann *gerichtet* und wird als *Digraph* benannt. Diese Bezeichnung ist von dem englischen Begriff "directed graph" abgeleitet. In Abbildung 2.7 ist ein solcher Graph zu sehen.

Multigraph, Schlinge:
Wenn Startknoten und Endknoten einer Kante gleich sind, so wird diese als *Schlinge* bezeichnet. Der dazugehörige Graph ist dann ein sogenannter *Multigraph* (siehe Abbildung 2.7). In diesem können auch parallele Kanten vorkommen.

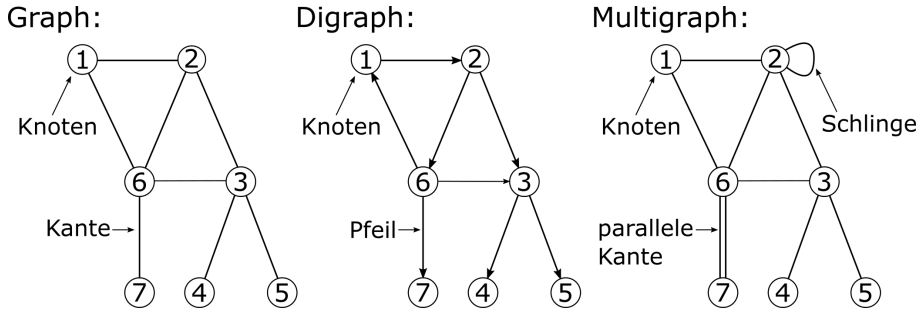


Abbildung 2.7: Graph, Digraph und Multigraph[2]

Baum, Wurzel, Blätter, Parent, Child:
Ein *Baum* ist ein spezieller Graphentyp. Dabei sind je zwei Knoten durch genau einen Weg miteinander verbunden. Die Knoten, welche den Abschluss bilden und nur über eine Kante verbunden sind, werden als Blätter bezeichnet. Wenn ein solcher Graph n Knoten besitzt, muss er über n-1 Kanten verfügen. Bäume können auch gerichtet sein. Dann wird ein Knoten, der vor einem anderen angeordnet ist, als *Parent* und der Folgeknoten als *Child* bezeichnet. Aufgrund des daraus resultierenden Aufbaus, gibt es genau einen Knoten, der kein Parent sondern nur Children besitzt. Dieser Knoten wird als Wurzel bezeichnet. Abbildung 2.8 zeigt einen Baum und seine Bestandteile. Der Knoten mit der Nummer "1" ist die Wurzel und der Parent von Knoten "2". Daher ist dieser Knoten das Child der Wurzel. Alle Knoten werden in Level eingeteilt, wobei sich die Wurzel in Level "0" befindet. Je weiter ein Knoten von der Wurzel entfernt ist, desto höher ist seine Level-Nummer.

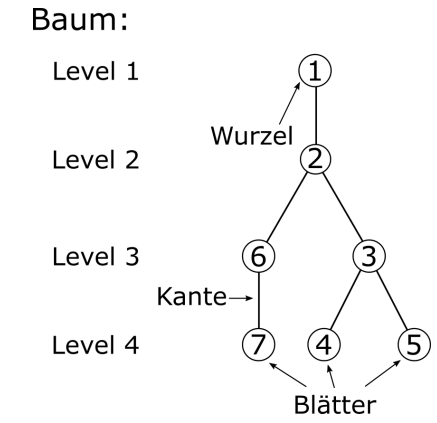


Abbildung 2.8: Baumgraph[2]

Isomorphie:
Zwei Graphen sind isomorph zueinander, wenn ihre Struktur gleich ist. Deren Darstellung kann aber voneinander abweichen. Mathematisch ausgedrückt sind zwei Graphen (G=(N,K) und G'=(N',K')) isomorph, wenn es eine bijektive Abbildung $\varphi : N -> N'$ gibt, sodass für alle $x, y \in N$ gilt:

$$[x, y] \in K \Leftrightarrow [\varphi(x), \varphi(y)] \in K'$$

Dabei ist N eine Menge von Knoten $N = \{n1, n2, ...\}$ und K eine Menge von Kanten $K = \{k1, k2, ...\}$ mit untergeordneten Kantenpaaren $k = \{x, y\}$. Dabei sind x und y $\in$ N. Der Graph ist beschrieben durch $G = (N, K)$. Abbildung 2.9 zeigt anhand von zwei unterschiedlich dargestellten Graphen ein Beispiel für Isomorphie. Auch die Bezeichnung der Knoten ist für den Isomorphie-Begriff vernachlässigbar. Die Knotenfarben erleichtern das Erkennen der Strukturgleichheit.

[2] Eigene Darstellung

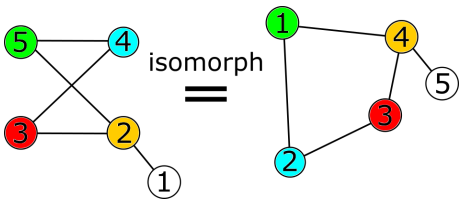


Abbildung 2.9: Isomorphie Beispiel[2]

2.2.2 Isomorphie von Bäumen bestimmen

Die Ermittlung der Isomorphie von Graphen ist komplex und nicht immer lösbar. Generische Graphen lassen sich aber in Bäume transformieren, deren Isomorphie immer feststellbar ist. Dabei muss die Umwandlung der zu untersuchenden Graphen nach dem gleichen Schema ablaufen. Zur Feststellung der Isomorphie ist unter [12] ein Algorithmus zu finden, der im Folgenden sinngemäß erörtert wird. In Abbildung 2.10 ist der Algorithmus grafisch dargestellt. Das Bild wurde dem Vorlesungsskript von Rolf Niedermeier [12] entnommen und leicht angepasst. Das Grundprinzip besteht darin, dass den Strukturen im Graphen Zahlen zugeordnet werden. Jede Struktur bekommt die gleiche Zahl, unabhängig davon zu welchem der beiden untersuchten Graphen diese gehört. Das heißt wenn eine Struktur im Baum T1 vorkommt, wird der gleichen Struktur im Graphen T2 die identische Zahl zugeordnet. Die Strukturen bestehen aus den Zahlen anderer Strukturen. Die kleinste Struktur sind die Blätter mit der Strukturzahl 0, bei deren Strukturzuordnung der Algorithmus beginnt. Wenn die Strukturzahlen der Wurzeln beider Bäume gleich sind, sind die Bäume isomorph. In der nachfolgenden Aufzählung sind die einzelnen Schritte des Algorithmus zusammen gefasst.

1. Bestimme die Blätter und ordne diese der Strukturzahl 0 zu.
2. Finde die Knoten, deren Children bereits allen eine Strukturzahl zugeordnet wurde.
3. Sortiere die Liste der Children-Strukturzahlen nach aufsteigender Reihenfolge. Existiert bereits eine Struktuzahl für eine strukturgleiche Liste?
   - Ja: Ordne dem Element die vorhandene Zahl zu.
   - Nein: Erzeuge eine neue Strukturzahl für diese Struktur.
4. War das letzte untersuchte Element die Wurzel?
   - Ja: Gehe zu Punkt 5.
   - Nein: Gehe zurück zu Punkt 2.

[2] Eigene Darstellung

5. Vergleiche die Strukturzahlen der Wurzeln. Sind diese gleich, so sind die Graphen isomorph.

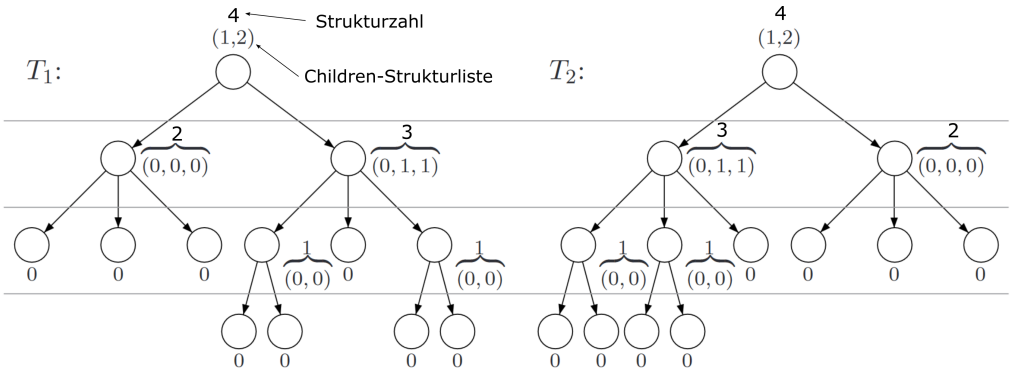


Abbildung 2.10: Isomorphiebestiummung von Bäumen[12]

2.2.3 Baum-Graphen in Tcl

Tcl ist eine Interpreter-Sprache, die in der Chip-Entwicklung weit verbreitet ist. Eine Ausführliche Dokumentation zur Sprache ist unter [3] zu finden. Zur Verwendung von Baum-Graphen in Tcl kann das Paket "struct::tree" genutzt werden. Die ausführliche Dokumentation dazu ist unter [10] zu finden. Das Paket verfügt über einige nützliche Funktionen und Algorithmen, um Bäume zu erzeugen und zu analysieren. Dazu wird ein Objekt instantiiert, über dessen Bezeichnung auf die Funktionen zugegriffen werden kann. Im Folgenden wird das Paket und seine Verwendung kurz erläutert.

`::struct::tree tree_name`:
Mithilfe dieses Kommandos kann ein neues Tree-Objekt mit der Bezeichnung `tree_name` erzeugt werden. Dieses Objekt wird einem globalen Tcl-Kommando zugeordnet, über dessen Argumente auf das Objekt zugegriffen werden kann.

`tree_name insert parent end child_name`:
Zum Einfügen von Knoten in den Graphen `tree_name` wird das `insert` Kommando verwendet. Als Argument wird die Bezeichnung des Parents übergeben, welchem der Knoten untergeordnet werden soll. Daraufhin erfolgt eine Angabe, an welcher Stelle der Children-Liste der neue Knoten eingefügt werden soll. Durch das Schlüsselwort "end" wird der neue Knoten an das Ende dieser Liste angehängt. Das letzte Argument ist die Bezeichnung des neuen Childs.

`tree_name set node key value`:
Der Befehl `set` ermöglicht es, einem Knoten Schlüsselwertpaare zuzuordnen. So kann über die Angabe eines Schlüssels auf einen Wert, der dem Knoten zugewiesen wurde, zugegriffen werden. Dazu wird die Bezeichnung des Knotens (`node`) ein Schlüsselwort (`key`) und ein Wert (`value`) übergeben.

`tree_name get node key`:
Mithilfe dieses Kommandos kann unter der Verwendung eines Schlüssels (`key`) der Wert, der einem Knoten (`node`) zugewiesen ist, abgerufen werden.

`tree_name keys node`:
Der Befehl `keys` gibt alle Schlüsselwertpaare als Liste zurück, welche dem gegebenen Knoten zugewiesen sind.

`tree_name depth node`:
Über das `depth`-Kommando kann die Tiefe eines Knotens ermittelt werden. Das bedeutet, dass ein Wert zurückgegeben wird, welcher der Anzahl der Knoten entspricht, die sich zwischen dem gegebenen Knoten und der Wurzel befinden.

`tree_name exists node`:
Um zu überprüfen, ob ein Knoten tatsächlich existiert, wird der Befehl `exists` verwendet. Dazu wird die Bezeichnung des Knotens als Argument übergeben.

`tree_name isleaf node`:
Mit dem Kommando `isleaf` kann festgestellt werden, ob ein Knoten ein Blatt ist. Als Argument wird die Bezeichnung des zu untersuchenden Knoten übergeben.

`tree_name leaves`:
Um eine Liste aller Blätter zu erhalten, welche dem Baum zugeordnet sind, wird der Befehl `leaves` ausgeführt.

`tree_name nodes`:
Dieses Kommando ist ähnlich wie der `leaves`-Befehl. Im Gegensatz dazu werden aber nicht die Blätter, sondern alle Knoten, die keine Blätter sind, als Liste zurückgegeben.

tree_name parent node:
Mithilfe des Kommandos `parent` kann der parent eines gegebenen Knotens ermittelt werden.

tree_name children ?-all? node ?filter cmdprefix?:
Mit diesem Befehl können die Children eines Knotens ermittelt werden. Durch das Schlüsselwort "-all" werden alle nachfolgenden Knoten zurückgegeben, also auch die Children der Children. Darüber hinaus kann über das Schlüsselwort "filter" ein Subkommando übergeben werden. Dieses Subkommando muss vorher als Prozedur initialisiert werden. Wenn es aufgerufen wird, werden automatisch die Graphen- und die Knoten-Bezeichnung als Argumente übergeben. Der Rückgabewert des Subkommandos muss ein boolscher Wert sein. Wenn der Rückgabewert "true" ist, wird der Knoten einer Liste hinzugefügt, die schlussendlich den Rückgabewert der gesamten Prozedur darstellt. Das folgende Listing ist [10] entnommen und zeigt wie das Subkommando funktioniert. Darüber hinaus ist zu sehen, wie Knoten einem Baum hinzugefügt werden.

```
1  mytree insert root end 0 ; mytree set 0 volume 30
2  mytree insert root end 1
3  mytree insert root end 2
4  mytree insert 0    end 3
5  mytree insert 0    end 4
6  mytree insert 4    end 5 ; mytree set 5 volume 50
7  mytree insert 4    end 6
8  proc vol {t n} {
9     $t keyexists $n volume
10 }
11 proc vgt40 {t n} {
12    if {![$t keyexists $n volume]} {return 0}
13    expr {[$t get $n volume] > 40}
14 }
15
16 tclsh> lsort [mytree children -all root filter vol]
17 0 5
18 tclsh> lsort [mytree children -all root filter vgt40]
19 5
20 tclsh> lsort [mytree children root filter vol]
21 0
22 tclsh> puts ([lsort [mytree children root filter vgt40]])
```

Quelltext 2.1: Subkommando: filter cmdprefix

tree_name size ?node?:
Dieser Befehl gibt die Anzahl aller nachfolgenden Knoten zurück. Wenn kein Knoten angegeben ist, wird die Wurzel als Standardknoten gesetzt.

`tree_name rename node new_name:`
Standardmäßig erhält der Wurzelknoten die Bezeichnung "root". Um die Bezeichnung zu ändern, kann das Kommando `rename` verwendet werden. Dazu wird die Bezeichnung "root" als Knoten übergeben. Es können auch die Bezeichnungen anderer Knoten geändert werden, indem die entsprechende Bezeichnung übergeben wird.

`tree_name rootname:`
Der Befehl `rootname` gibt die Bezeichnung des Knotens root zurück.

Es existieren noch weitere Kommandos, mit denen Graphen untersucht werden können. Dabei werden auch Algorithmen wie der *Breath-First-Algorithmus* oder der *Deapth-First-Algorithmus* verwendet. Diese werden jedoch nicht in dieser Masterthesis benötigt. Bei Interesse sind weitere Informationen unter [10] zu finden.

2.3 NLView

NLView ist eine Graphical-User-Interface (GUI) Komponente, mit der es möglich ist, *Schematics* zu generieren und anzuzeigen. Die Komponente ist in verschiedenen Softwarepaketen, deren Ausführungen sich in der verwendeten Programmiersprache unterscheiden, verfügbar. Die ausführliche Dokumentation ist unter [4] zu finden. In diesem Projekt wird die Qt Ausführung verwendet. Diese Version verfügt über eine Demo-Applikation, auf die der *Clocktree-Analyser* aufbauen soll. Weitere Informationen zu Qt können [5] entnommen werden. Um auf die Algorithmen und die Funktionalität von *NLView* zugreifen zu können, bietet der Entwickler *Concept Engineering* eine Application Programming Interface (API) Schnittstelle an. Abbildung 2.11 zeigt den Aufbau von *NlView* unter Berücksichtigung der verschiedenen zur Verfügung gestellten Schnittstellen. Diese werden im Rahmen dieses Projektes jedoch nicht verwendet und finden daher auch keine weitere Beachtung. Die rot-gestrichelte Linie umfasst alle Komponenten der Engine, die im Rahmen des Projektes verwendet werden. Dazu gehören eine benutzerspezifische Applikation (Customer App), die API-Schnittstelle (Core API), interne Algorithmen, ein *GUI Wrapper* und eine GUI Applikation. *NLView* kann als Qt-Klasse mit der Bezeichnung *NlvQWidget* in die benutzerspezifische Anwendung eingebunden werden.

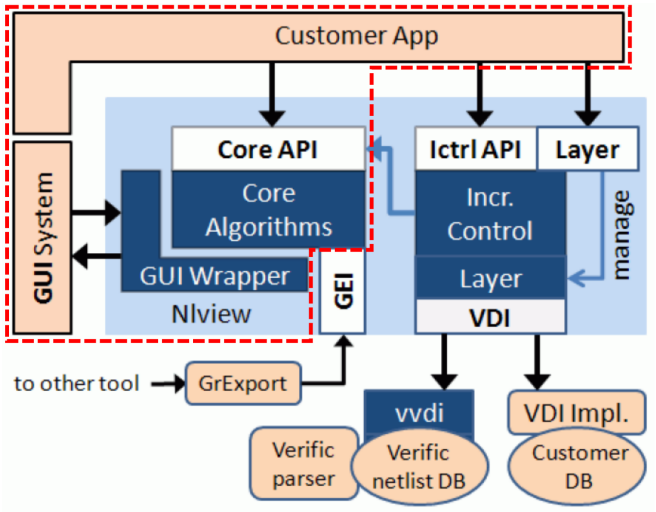


Abbildung 2.11: NLView Überblick[4]

2.3.1 API-Kommandos

Die API-Kommandos können entweder als Argumente an die Memberfunction *commandLine()* der Qt-Klasse *NlvQWidget* übergeben oder als Dateien eingebunden werden. Bei der zweiten Variante ist es möglich, mehrere Kommandos zu verwenden und so vollständige *Schematics* einzulesen. Auch für das Einlesen existiert ein Kommando. In diesem Abschnitt werden die für das Projekt verwendeten API-Kommandos erläutert. Weitere Informationen sind in der *NLViewQT Documentation* zu finden.

`property name value`:
Mit dem Kommando `property` können Anzeigeoptionen wie Hintergrundfarben oder Schattierungen festgelegt werden. Dazu wird der Name der Option und ein Wert als Argument übergeben. Eine Auflistung aller Attribute ist der *NLViewQT Documentation* zu entnehmen. Die Anzeigeoption `decorate` hat jedoch eine gesonderte Bedeutung. Mit ihr ist es möglich, Buttons an selektierten Objekten anzuzeigen, welche mithilfe des `bind` Kommandos eine Funktion erhalten. Abbildung 2.12 zeigt die Buttons an einer Hierarchie, die sich auffalten und zusammenfalten lässt.

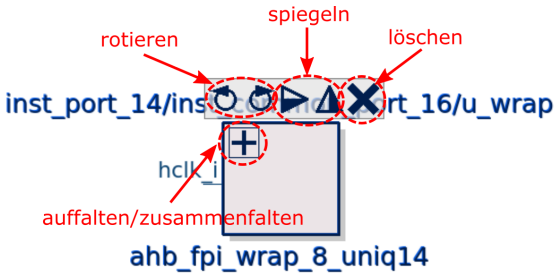


Abbildung 2.12: Decorate Button

`bind type -action action ?action_options?:`
Mit Hilfe des `bind` Kommandos ist es möglich, Maus-Events (*mousebindings*) in *NLView* einzubinden und so mit dem *Schematic* zu interagieren. Das Kommando ist wie folgt aufgebaut: Das Schlüsselwort `type` setzt sich aus den Schlüsselwörtern `(<modifiers>-)<event>-<button>(-<object>)` zusammen. So lassen sich über den Eintrag `modifiers` Modifikatoren wie *Shift, Alt* und *Strg* ergänzen. Das Wort `event` wird durch das Maus-Event ersetzt, welches besagt, bei welcher Maus-Interaktion etwas geschehen soll. Dazu gehören *click, doubleclick, stroke* und *wheel*. Daraufhin wird die Taste angegeben, auf der die Interaktion stattfindet. Diese Angabe wird durch eine Zahl interpretiert. `1` steht für die linke Maustaste, `2` für das Mausrad und `3` für die rechte Maustaste. Über die Option `object` kann das *mousebinding* auf bestimmte Schematic-Elemente beschränkt werden. Dazu gehören nicht nur Teile der Schaltung, sondern auch im Schematic implementierte *Buttons*. Anschließend wird eine vordefinierte Aktion für das *mousebinding* angegeben. Dazu stellt *NLView* eine Auswahl zur Verfügung. Die folgende Auflistung erwähnt die wichtigsten Aktionen:

- **mirror_y:** Spiegelung eines Objektes über die Y-Achse
- **mirror_x:** Spiegelung eines Objektes über die X-Achse
- **hier_unfold:** Auffalten von Hierarchien
- **hier_fold:** Einfalten von Hierarchien
- **rotate_right:** Rotieren eines Objektes nach rechts
- **rotate_left:** Rotieren eines Objektes nach links
- **remove:** Löschen eines Objektes

Alle oben erwähnten Aktionen können noch durch ein *iflag* erweitert werden. Damit ist es möglich, das angezeigt Schematic an die durch das *mousebinding* verursachte Änderung anzupassen. Ein Beispiel ist die Neuanordnung aller Objekte, nachdem eines gelöscht wurde. Über das letzte Schlüsselwort können Textausgaben gemacht werden. So ist es möglich, über einen Button die

Funktionalität des *mousebindings* anzuzeigen, wenn der Mauszeiger sich darüber befindet. Im folgenden Listing ist zu sehen, wie Objekte gelöscht werden können. Dazu wird der *remove-Button* eines Objektes und ein *mousebinding* verwendet. Befindet sich der Mauszeiger auf dem Button, so wird der Text "Löschen" angezeigt.

```
1 bind "click-1-button @remove" -action remove -echo_text Löschen
```

Quelltext 2.2: bind

`module command ?modname viewname?:`
Ein *NLView*-Wiget verfügt über mehrere Module, wovon jedes wiederum aus Schematic-Elementen wie: Instanzen (Flipflops, Buffern, usw.), Symbolen, Netzen, Ports und ähnlichen Objekten besteht. Es kann immer nur ein Modul angezeigt werden. Dieses wird als *current module* bezeichnet. Dem Kommando `modul` können folgende Subkommandos übergeben werden:

- **new modname ?vname?:** Mit diesem Befehl wird ein neues Modul angelegt. Es kann ein beliebiger Modulname angegeben werden. Zusätzlich kann auch ein optionaler *Viewname* gesetzt werden. Wird ein Modulname ein zweites mal vergeben, so wird das erste Modul überschrieben. Wird jedoch dabei ein anderer *Viewname* vergeben, dann bleiben beide Module erhalten.
- **open modname ?vname?:** Mit diesem Kommando können Module geöffnet und angezeigt werden. Diese sind dann als *current module* gesetzt.
- **delete modname ?vname?:** Zum Löschen eines Moduls wird dieses Kommando und der entsprechende Modul- und gegebenenfalls *Viewname* übergeben.
- **info:** Um den Modul- und *Viewnamen* des *current module* zu erhalten, kann dieser Befehl verwendet werden.

`load item options:`
Um ein Schematic darstellen zu können, muss ein Modul geladen werden, das über eine Datenbasis verfügt. Mit dem `load` Kommando können Objekttypen wie Instanzen, Netze, oder Ports in diese Datenbasis geladen werden. Damit es möglich ist Instanzen darzustellen, müssen diese mit Symbolen hinterlegt sein. Instanzen können unterschiedliche logische Elemente wie Flipflops, Inverter oder andere Logik-Gatter sein. Die Symbole müssen ebenfalls in die Datenbasis geladen werden. Indem hinter dem `load` Kommando der Objekttyp (`item`) und dahinter einige Eigenschaften (`options`) angegeben werden, können die unterschiedlichen Schematic-Objekte dem Modul hinzugefügt werden. Die Objekte werden dem *current module* zugeteilt. Im folgenden sind die relevanten Objekttypen und deren Eigenschaften aufgelistet.

- **symbol name viewname stype pins ?options?:** Mit diesem Kommando können Symbole in die Datenbasis geladen werden. Dazu wird die Symbolbezeichnung, über die später auf

das Symbol zugegriffen werden kann, und der optionale *Viewnamen* des Moduls angegeben. Daraufhin folgt eine Zelldefinition (`stype`), welche die Form des Symboles beschreibt. *NLView* gibt einige Definitionen, wie Logische-Gatter und Multiplexer, vor. Zur Beschreibung rechteckiger Formen wird der `stype` "BOX" angegeben. Hierarchische Elemente werden mit "HIERBOX" kategorisiert. Die gesamte Auswahl ist der *NLViewQT Documentation* zu entnehmen. Anschließend folgt die Definition der Pins. Im folgenden Listing wird ein Buffer-Symbol erzeugt.

```
1   load symbol R10070L_USBUFX018 NLExportView BUF port A input.left port Z
        output.right
```

Quelltext 2.3: load symbol

- **inst name symbolname viewname ?options?:** Um Objekte zu erzeugen, wird nach dem `load` Befehl das Schlüsselwort `inst` angegeben. Daraufhin folgt die Bezeichnung der Instanz (`name`) und der Symbolname (`symbolname`), welcher die Darstellung des Objektes beschreibt. Daraufhin kann optional der *Viewname* des Moduls eingetragen werden. Anschließend können, durch die Angabe bestimmter Optionen, Attribute wie Koordinaten oder Farben angepasst werden. Über das Schlüsselwort `-hier` ist es möglich, Instanzen einer Hierarchie zuzuweisen. Außerdem kann über die *Flags*: `-fold` und `-unfold` angegeben werden, ob eine Hierarchie gefaltet sein soll. Das Listing zeigt die Instanziierung einer Hierarchie und eines weiteren Elements, das sich innerhalb der Hierarchie befindet.

```
1   load inst m6332 m6332 NLExportView -fold
2   load inst inst_ip_canfd0 ip_mcan_ahb_0_0_uniq0 NLExportView -hier m6332
```

Quelltext 2.4: load inst

- **port name direction ?options?:** Für das Hinzufügen von Ports wird der Port-Name (`name`) und die Richtung des Ports benötigt. Am Ende des Kommandos können Attribute wie Farben oder Koordinaten angegeben werden. Das folgende Listing zeigt die Instanziierung eines Ports.

```
1   load port sx_clocks_cpu_pfi_pfrwb_clks_max_i in 2
```

Quelltext 2.5: load port

- **net name connections ?options?:** Zur Instanziierung von Netzen wird der Netz-Name (`name`) und eine Liste der Verbindungen benötigt. Ein Element dieser Liste wird mit dem Schlüsselwort `-pin` eingeleitet. Daraufhin folgt der Name einer Instanz und die entsprechende Pin-Bezeichnung. Hierarchien verfügen über zwei Möglichkeiten der Pin-Angabe. Die erste ist wie bei normalen Instanzen und verbindet den äußeren Teil eines Pins mit einem Element. Die zweite Variante verbindet den inneren Teil eines Pins und wird mit dem

Schlüsselwort `-hierPin` eingeleitet. Abbildung 2.13 zeigt den Unterschied zwischen den beiden Pin-Typen. Es ist zu beachten, dass beide Pins selektiert und damit hervorgehoben sind.

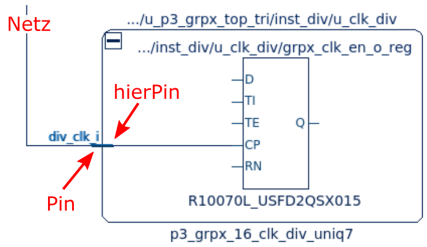


Abbildung 2.13: Unterschied zwischen Pin und HierPin

Ports können über das Schlüsselwort `-port` und dem Port-Namen in die Kontaktliste einbezogen werden. Anschließend können Attribute wie Farbe und Linienbreite festgelegt werden. Im folgenden Listing wird die Instanziierung eines Netzes gezeigt.

```
1   load net inst_port_03/pad_tck_i -pin inst_port_03/inst_common_port_12
        pad_tck_i -hierPin inst_port_03 pad_tck_i
```

Quelltext 2.6: load net

`increment ?options?`:
Wenn ein Schematic bereits generiert und im Widget angezeigt wird, müssen inkrementelle Veränderungen der Darstellung in einem *increment/show-cycle* stattfinden. Zu den Änderungen, die eine Aktualisierung erforderlich machen, gehört das Laden neuer Instanzen, Rotieren und Entfernen von Objekten sowie das Falten von Hierarchien. Der Prozess wird mit dem Kommando `increment` eingeleitet und kann mit verschiedenen Optionen versehen werden. Die wichtigste Option ist `-optimize`, mit der nach dem Prozess alle Objekte im Schematic geordnet dargestellt werden. Der *increment/show-cycle* wird mit dem Schlüsselwort `show` abgeschlossen, woraufhin die Veränderungen im Schematic angezeigt werden.

`show`:
Dieses Kommando dient dazu den *increment/show-cycle* zu beenden. Außerdem kann so die Erzeugung eines neuen Moduls abgeschlossen werden. Nachdem das Kommando `module new modname` eingegeben wurde, befindet sich die Engine im inkrementellen Modus. So kann direkt nach dem Befehl eine Datenbasis erzeugt werden. Auch dieser Modus muss mit dem `show` Kommando verlassen werden.

`fullfit`:
Dieses Kommando passt das angezeigte Schematic an die Größe des Widgets an.

`source ?-string? text|filename`:
Mit dem `source` Kommando können Dateien oder Strings in *NLView* eingebunden werden. Für die Einbindung von Dateien wird der Pfad und der Dateiname angegeben. Um eine Zeichenkette einzubinden, werden das Schlüsselwort `-string` und anschließend die Zeichenkette angegeben. Innerhalb dieser Zeichenkette müssen die einzelnen Kommandos durch Zeilenumbrüche ("\n") getrennt sein.

`selection ?-append OID?`:
Die Referenzierung der Objekte findet durch sogenannte Object Identifiers (OIDs) statt. Diese bestehen mindestens aus dem Objekttypen und der Objektbezeichnung. Die Identifiers (IDs) der Netze bestehen nur aus diesen beiden Elementen. Instanz-IDs beinhalten zusätzlich den Symbolnamen und den Modul-Viewnamen. Pins hingegen bestehen aus dem Namen der Instanz und der Bezeichnung des Pins sowie der Richtung des Pins. Die OID des gerade selektierten Objektes kann, mithilfe des Kommandos `selection`, ermittelt werden. Durch das Schlüsselwort `-append` und der Angabe einer OID, können Objekte der Selektions-Liste angeheftet werden.

`attribute OID -attr attrname value`:
In *NLView* werden *Attribute* verwendet, um die Eigenschaften eines Objektes in dem Schematic zu manipulieren. So können die Farbe oder die sichtbare Bezeichnung eines Objektes angepasst werden. Da nicht die Schaltungsstruktur des aktuell angezeigten Schematics verändert wird, ist zum Setzen von Attributen kein *increment/show-cycle* erforderlich. Dazu wird zunächst nach dem Kommando `attribute` die OID des Objektes angegeben. Anschließend folgt das Schlüsselwort `-attr`, der Name des Attributes und ein Wert, der dem Attribut zugeteilt werden soll. Prinzipiell kann ein Attribut jede beliebige Bezeichnung tragen. *NLView* gibt an dieser Stelle einige Attributbezeichnungen vor, die einen Effekt in der Darstellung auslösen. Sie beginnen alle mit einem "@" und können in der *NLViewQT Documentation* nachgeschlagen werden. Das wichtigste Attribut ist dabei das *@color*-Attribut. Mit diesem kann die Farbe der Instanzen und Netze angepasst werden. Als Wert wird eine Farbe im Hexadezimal-Format, mit 12, 24, 36 oder 48 Bit Farbtiefe, angegeben. Attribute können auch im Zusammenhang mit dem `load` Kommando verwendet werden. Dazu ist die Angabe des Schlüsselwortes `-attr`, der Attribut-Name und der Wert erforderlich. Abbildung 2.14 zeigt die Funktionsweise der beiden wichtigsten Attribute. Im unteren Teil der Abbildung sind auch die erforderlichen Kommandos zu sehen.

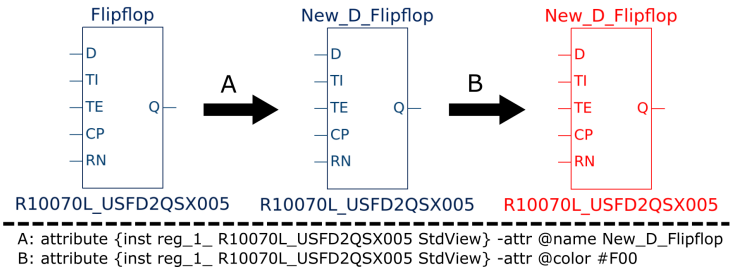


Abbildung 2.14: Die Attribute @name und @color

`flag OID option`:

Mit dem `flag` Kommando ist es möglich, die Darstellungsart der Objekte anzupassen. Dazu gehört die Bewegung und der Objekte sowie die Veränderung der Größe sowie die Faltung von Hierarchien. Dies stellt eine Änderung des Schematics dar und muss daher in einem *increment/show-cycle* ausgeführt werden. *Flags* können an das `load` Kommando angehängt oder über den Befehl `flag` gesetzt werden. Eine Liste aller möglichen Flags ist in der *NLViewQT Documentation* zu finden. Die wichtigsten sind `-fold`, `-unfold`, `-autohide` und `-unautohide`. Mit den ersten beiden können Hierarchien gefaltet, mit den letzten ungenutzte Pins eines Objektes ausgeblendet werden. In der Abbildung 2.15 sind die Auswirkungen dieser Flags dargestellt.

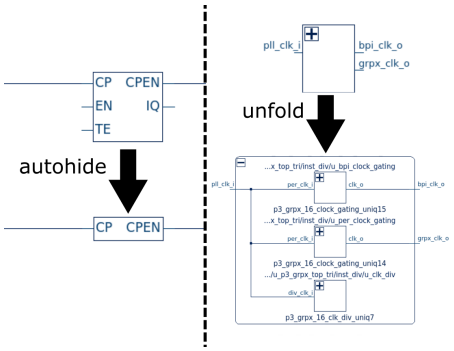


Abbildung 2.15: Autohide und unfold

`save ?optiones? filename`:

Zur Speicherung des aktuellen Standes von *NLView* kann der `save` Befehl verwendet werden. Dabei werden alle Daten in einer Datei gespeichert, deren Name am Ende des Kommandos angegeben wird. Über optionale Schlüsselwörter ist es möglich, das *current module* zu speichern. Ohne Verwendung dieser Schlüsselwörter werden alle Module gespeichert. Weitere Schlüsselwörter können aus *NLViewQT Documentation* entnommen werden.

2.3.2 Demo-Applikation

Im Softwarepaket von *NLView* ist die GUI unabhängige *NLView-core* Bibliothek enthalten. Die Formatierung der Bibliothek basiert auf einer Maschinensprache. Zudem ist ein *Wrapper* im Qt-Quellcode Format mit Header- ("wrapper.h") und Source-Datei ("wrapper.cpp") enthalten. Im Code wird die Klasse `NlvQWidget` deklariert. Diese erbt von der Qt Basisklasse `QWidget` und bindet die *NLView-core* Bibliothek sowie das *Qt-Renderinterface*, welches ebenfalls in einer Header ("nlvrqt.h") und einer Source-Datei ("nlvrqt.cpp") vorliegt, ein. Das *Qt-Renderinterface* verfügt über Funktionen, mit denen es möglich ist, die Schematics auf einem *QPaintDevise* darzustellen. Dazu gehören Bildschirmausgaben oder Drucker. Abbildung 2.16 zeigt die Verknüpfungen der einzelnen Komponenten. In grün ist dabei die benutzerspezifische Applikation, welche im Bild dem Demo Programm entspricht, dargestellt. Der *Wrapper* bindet Funktionen von *Qt-Renderinterface* und *NLView-core* ein. In der Demo Applikation wird ein Objekt der Klasse `NlvQWidget` erzeugt, mit deren Memberfunctions die *NLView-Engine* verwendet werden kann.

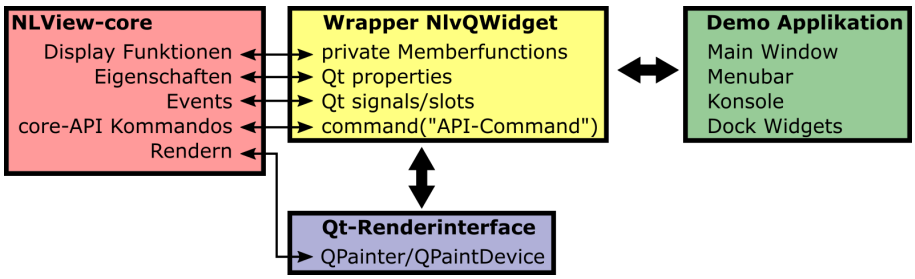


Abbildung 2.16: Das *NLView*-Wrapper Interface

Die *NLView* Demo Applikation besteht aus den Klassen `Demo`, `MyComboBox` und aus der von `NlvQWidget` abgeleiteten Klasse `Nlview`. Abbildung 2.17 zeigt die Klasse `NLView`. Im Bild sind die wichtigsten Funktionen und Zusammenhänge von *Signalen* und *Slots* erläutert. Die Signale werden hauptsächlich durch Maus- und Tastaturevents ausgelöst. Innerhalb der dadurch aktivierten Slots werden dann entsprechende API-Kommandos ausgeführt, welche als Argument an die Memberfunction `commandLine()` der Klasse `NLvQWidget` übergeben werden.

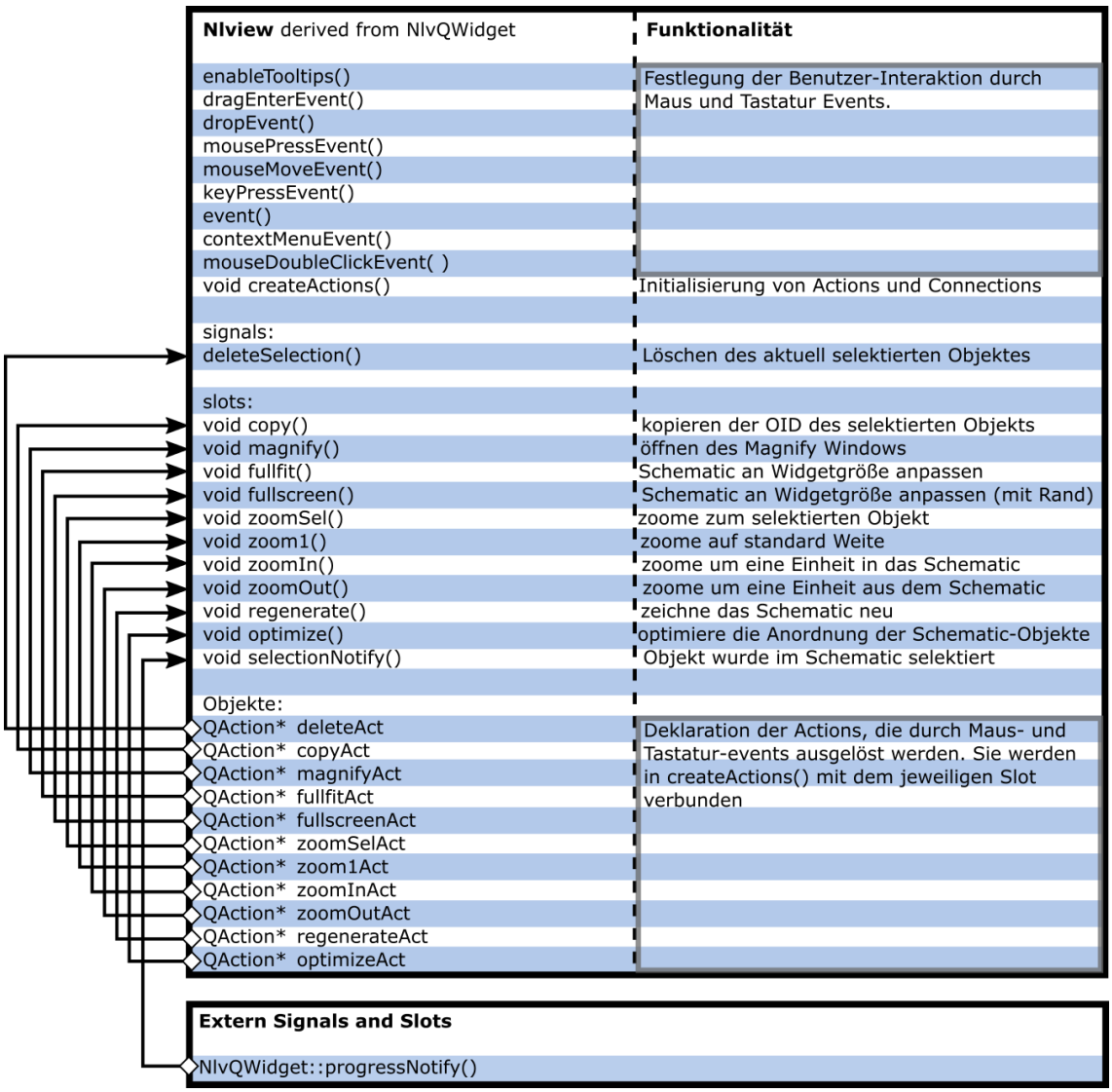


Abbildung 2.17: Die Klasse `NLView`

Die Klasse `Nlview` wird in der `Demo`-Klasse, durch die Erzeugung eines Objektes, eingebunden. Zudem wird ein Objekt der Klasse `MyComboBox` instanziiert. Dieses wird als Kommandokonsole verwendet und verfügt über spezielle Eigenschaften, wie etwa die Ausführung von Kommandos bei Betätigung der Enter-Taste oder dem Wiederaufruf bereits verwendeter Kommandos durch die obere Pfeiltaste. Dazu wird ein klasseneigenes Signal erzeugt, das solche Tastatureingaben auswertet. Die Klasse *Demo* verfügt auch über eine Menüleiste, in der einige *Actions* erzeugt werden. Dazu gehört das Öffnen von NLV-Dateien, das Speichern des aktuellen Schematics in einem gängigen Bildformat oder das Öffnen einer *Minimap*, welche den Überblick über das angezeigte Schematic verbessern soll. In Abbildung 2.18 sind die Klassen *Demo* und *MyComboBox* erläutert. Außerdem sind deren Zusammenhänge und der Zusammenhang der Klasse *Nlview*, durch *Slots*

und *Signale* dargestellt. Darüber hinaus bietet die Beschreibung der wichtigsten Memberfunctions ein grundlegendes Verständnis über die Klasse *Demo*.

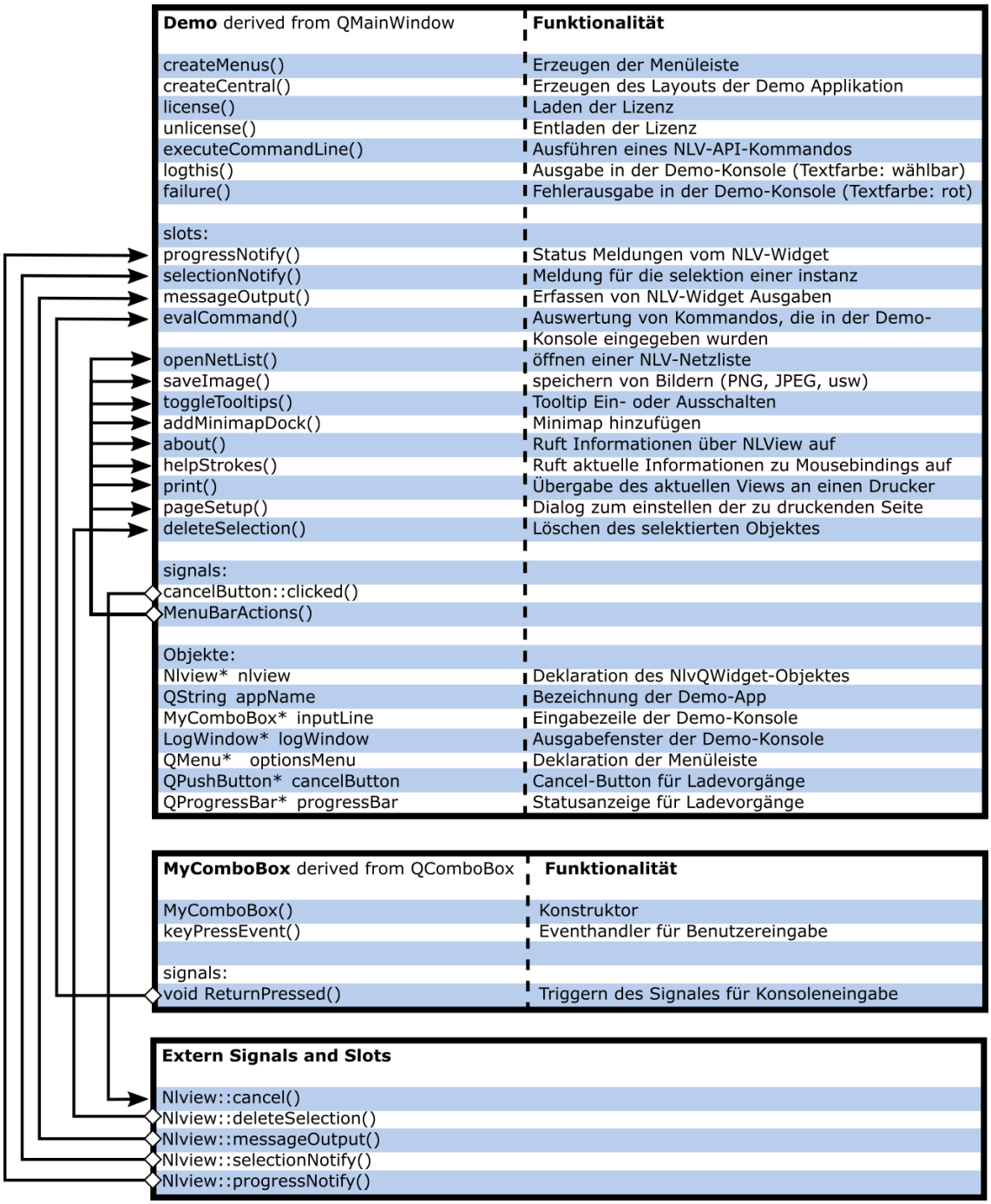


Abbildung 2.18: Die Klassen `Demo` und *MyComboBox*

In Abbildung 2.19 ist die GUI der Demo-Applikation zu sehen. Dort sind einige Bereiche mit roter Schrift gekennzeichnet. Wichtige Bestandteile der Applikation sind die Menüleiste, das

Schematic Widget, das Rechtsklick-Menü, die Konsole und die *Minimap*. In die Konsole können API-Kommandos eingegeben werden. Daraus resultierende Ausgaben erscheinen im Konsolen-Ausgabefenster. Einige Elemente der Demo-Applikation, wie das *Ictrl*-Interface, sind für dieses Projekt nicht relevant und sind daher nicht weiter erläutert.

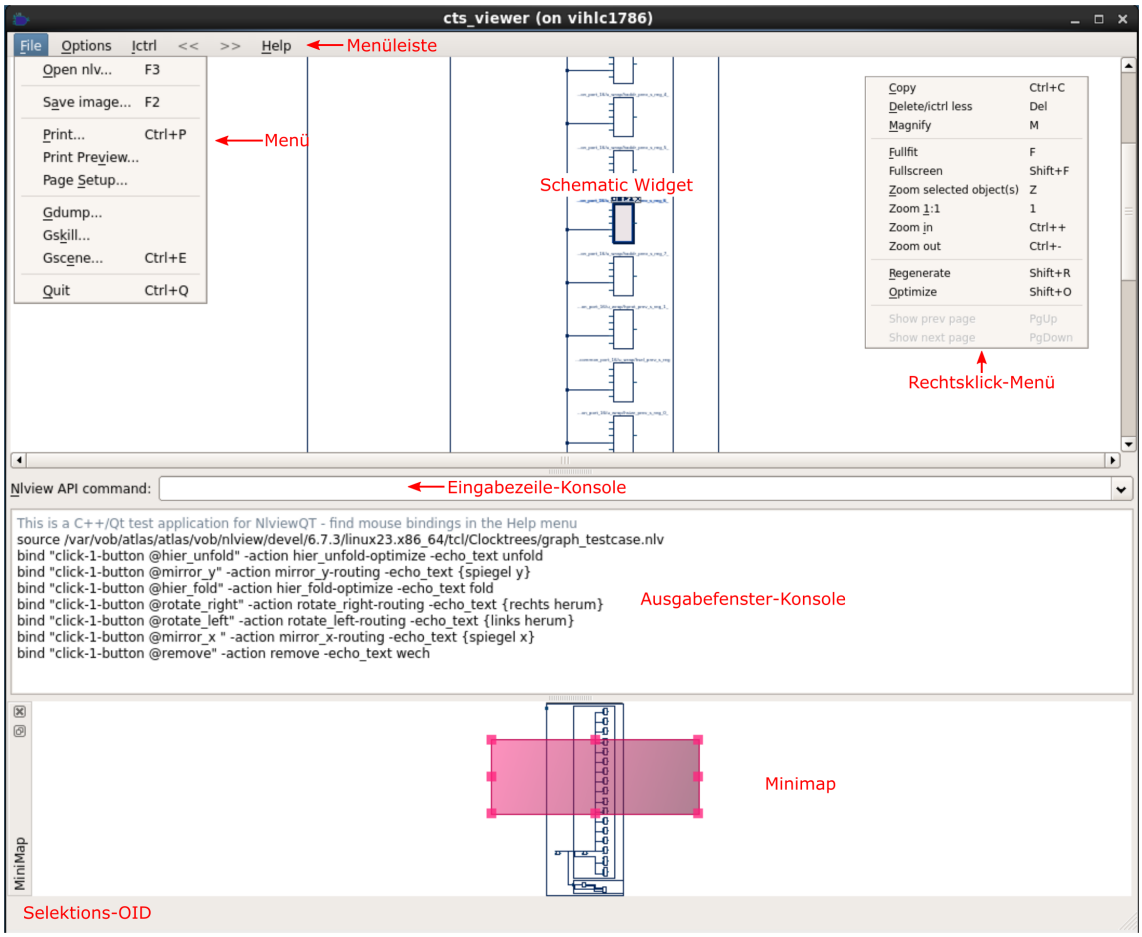


Abbildung 2.19: GUI der Demo-Applikation

3 Das bestehende Clocktree-Analyse-Tool

In diesem Kapitel wird das bestehende Clocktree-Analyse-Tool, welches als *Clocktree-Analyser* bezeichnet wird, beschrieben. Die Software besteht aus einer Qt-Anwendungen, welche die NLView-Engine einbindet, und einer Tcl-Applikation, die eine interne Datenbasis beinhaltet. Zudem verfügt die Tcl-Applikation über Algorithmen zum Einpflegen und weiteren Verarbeiten von Datensätzen. Abbildung 3.1 zeigt den Aufbau des *Clocktree-Analysers* und die vorhandenen Schnittstellen zwischen den Anwendungen. Da die Tcl-Applikation von der Qt-Anwendung als QProcess gestartet wird, können die Standard-Datenströme stdin, stdout, und stderr der Tcl-Application für Standardausgaben verwendet werden. Dazu zählen auch Fehlerausgaben, die in der Qt-Anwendung ausgegeben werden können. Der große Nachteil der Standard-Datenströme gegenüber einer Transmission Control Protocol/Internet Protocol (TCP/IP) Verbindung ist die geringe Geschwindigkeit. Daher werden große Datenmengen über die TCP/IP Schnittstelle übertragen. Der *Clocktree-Analyser* verfügt auch über eine Tcl-Konsole. Die Ein- und Ausgabe der Konsole befindet sich in der Qt-Anwendung und die Verarbeitung der Kommandos, die ebenfalls per TCP/IP übertragen werden, werden von der Tcl-Applikation verarbeitet und über die Standardausgaben zurückgeführt.

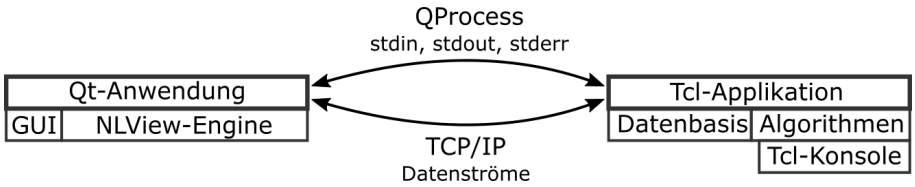


Abbildung 3.1: Architektur des Clocktree-Analysers

Die Dateien, welche die Daten zur Darstellung eines Clocktrees enthalten, basieren auf den API-Kommandos der *NLV-Engine*. Sie haben die Dateiendung ".nlv" und können direkt in der NLView Demo-Anwendung eingelesen werden. Zudem werden sie auch von der Tcl-Applikation eingelesen, um so die interne Datenbasis des *Clocktree-Analysers* zu erzeugen. Die Formatierung der Dateien ist in dem folgenden Listing aufgeführt. An oberster Stelle stehen die *Mausbindings*, gefolgt von den *Properties*. Anschließend wird ein Modul erzeugt und die Schematic-Objekte instanziiert. Daraufhin werden *authide*-Flags gesetzt, um *Macros* mit zahlreichen ungenutzten Pins übersichtlicher zu gestalten. Zum Schluss wird der Clocktree angezeigt.

```
1 bind "click-1-button @mirror_y" -action mirror_y-routing -echo_text {spiegel y}
```

```
...
property decorate 15
...
module new m6332 NLEView
load symbol cm4_e NLEView HIERBOX port hclk input.left
...
load inst inst_ip_0 cm4_e NLEView -fold -hier m6332 -attr @name inst_opamp0
...
load net ip_cluster0_sig -pin inst_scu0 clk_i -pin inst_cluster0 clk_o
...
flag {inst inst_ip_0} -autohide
...
show
fullfit
```

Quelltext 3.1: .nlv-Format

3.1 Tcl-Applikation

Die Verwendung einer Tcl-Applikation bietet den Vorteil, dass Tcl als Standardsprache in der SoC Entwicklung etabliert ist. So können beispielsweise einige Funktionen der Anwendung, die ohne GUI-Elemente auskommen, in andere Tools, welche über eine Tcl-Konsole verfügen, eingebunden werden. Die Tcl-Applikation besteht aus den vier Datein: *CtaMain.tcl*, *CtaMainProcs.tcl*, *CtaSubProcs.tcl* und *CtaNlvProcs.tcl*. Alle Dateien werden in einer Tcl-Shell eingebunden, die von der Qt-Anwendung als QProcess gestartet wird. *CtaMainProcs.tcl* beinhaltet Prozeduren für einige Algorithmen wie der Komprimierung der Schaltung oder der Separation von Schaltungsteilen. Die Datei *CtaSubProcs.tcl* beinhaltet Prozeduren, die nicht für die Algorithmen spezifiziert sind. Das umfasst den Umgang mit Listen und Arrays oder ähnliche Aufgaben. In dem Dokument *CtaNlvProcs.tcl* sind die abgebildeten API-Kommandos hinterlegt. Die Datei *CtaMain.tcl* beinhaltet die Kommandos zur Konfiguration der TCP/IP Schnittstelle und umfasst die Hauptfunktionalität der Tcl-Applikation. Abbildung 3.2 stellt diese Funktionalität anhand eines Diagrammes dar. Zu Beginn des Programmstarts wird ein *Socket-Server* mit der Port-Nummer "49155" erzeugt. Ist die Instanziierung des Ports nicht erfolgreich, so wird die Port-Nummer inkrementiert und ein neuer Versuch für die Erzeugung eines Servers unternommen. Bei erfolgreicher Instanziierung des Servers wird die Port-Nummer über den Standard-Datenstrom an die Qt-Applikation transferiert. Anschließend wartet das Programm auf die Initialisierung der Variablen mit der Bezeichnung "forever". Diese wird erst dann initialisiert, wenn das Programm beendet werden soll. Sobald ein Datensatz über die Verbindung empfangen wird, bricht die Applikation den Wartezustand ab und beginnt die Auswertung des Datensatzes. Eine solche Unterbrechung des Programms wird als *Interrupt* bezeichnet. In diesem Fall ist der unterbrechende Mechanismus ein TCP/IP-Socket. Daher

wird von einem *Socket-Interrupt* gesprochen. Ein derartiger initialer Datensatz beginnt mit einem CTA-Kommando, gefolgt von dem Modul- und dem Viewnamen sowie optionalen Argumenten. Innerhalb der Prozedur "accept" wird das Kommando ausgewertet. Anschließend beginnt der Datenaustausch. Sobald die Übertragung beendet ist, wird erneut auf die Initialisierung der Variablen "forever" gewartet. Durch die Ausführung eines bestimmten CTA-Kommandos wird die Variable "forever" initialisiert und so die Tcl-Shell beendet.

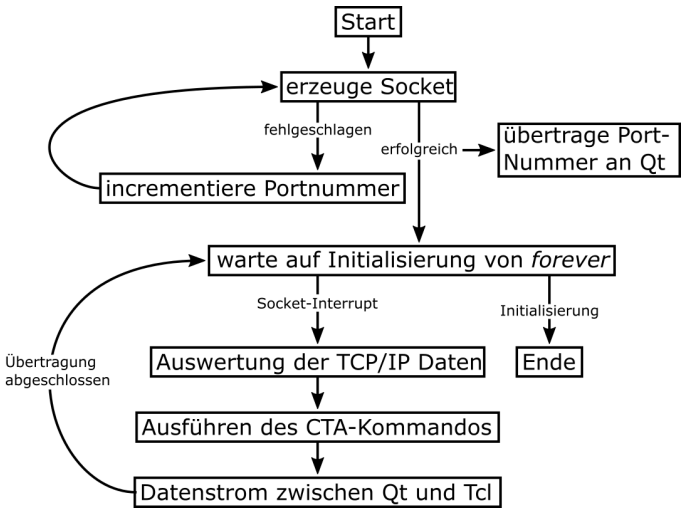


Abbildung 3.2: Funktionsweise der Tcl-Main-Anwendung

Der zentrale Kern der Applikation ist die Auswertung der CTA-Kommandos. Hier wird die Datenbasis eingelesen, erweitert, manipuliert und das Programm beendet. In Tabelle 3.1 sind die wichtigsten Kommandos und deren Funktionalität aufgeführt.

Tabelle 3.1: CTA-Kommandos

CTA-Kommando	Funktion
create_namespace	Erzeuge eine neue Datenbasis durch das Einlesen einer .nlv Datei.
exit	Beende die Applikation.
tcl	Führe ein Tcl-Kommando aus
group	Gruppiere Flipflops und Clockgates
select_part	Separiere die selektierte Hierarchie als neues Modul

3.1.1 Tcl-Datenbasis

Die interne Datenbasis der Tcl-Applikation befindet sich in Namespaces. Namespaces (deutsch: Namensräume) sind Konstrukte aus der Programmierung, die den Überblick über ein Programm

verbessern und Konflikte durch gleiche Bezeichnungen von Variablen verhindern. Solche Namensräume sind hierarchisch aufgebaut, sodass auf einzelne Elemente in den Namespaces über Namespace-Pfade zugegriffen werden kann. In tcl werden Namensräume durch zwei Doppelpunkte voneinander getrennt. Es ist nicht nur die Zuweisung von Variablen in Namespaces möglich, sondern auch die Zuweisung von Prozeduren. Im folgenden Quellcode wird die Erzeugung von zwei Namespaces gezeigt. Die innerhalb der Namespaces deklarierten Variablen haben die gleiche Bezeichnung, können aber durch die Referenzierung, der unterschiedlichen Namensräume, unterschiedliche Werte annehmen.

```
1 namespace eval namesp_A {set var 1}
2 namespace eval namesp_B {set var 2}
3 puts $namesp_A::var ;# -> 1
4 puts $namesp_B::var ;# -> 2
```

Quelltext 3.2: Namespace

Für jedes Modul (bzw. jeden Clocktree), der in den CTA eingelesen wird, wird ein eigener Namespace erzeugt. Dessen Bezeichnung besteht aus dem Modulnamen, gefolgt von einem Unterstrich und dem Viewnamen. Jeder Namespace verfügt über Arrays, die den Inhalt der NLV-Dateien sowie Informationen und Zusammenhänge, die aus den Dateien abgeleitet werden, enthalten. In der Tabelle 3.2 sind die Arrays und deren Inhalt aufgelistet. Dort sind auch Inhalte aufgeführt, die bereits bei der Planung der Datenbasis berücksichtigt wurden, aber erst im Rahmen dieser Masterthesis umgesetzt werden. Beispielsweise gilt es beim Array "NET" zu beachten, dass die Netze durch die Pins der Hierarchien getrennt sind. Diese Netze, die eigentlich verbunden sind, sollen in Netzgruppen gespeichert werden. Mit Hilfe dieser Netzgruppen soll die Struktur des Clocktrees ohne Hierarchien ermittelt werden können. Im Rahmen eines Kompressionsalgorithmuses werden Hierarchien erzeugt, welche Elemente enthalten, die als ein Objekt zusammengefasst werden sollen. Diese Hierarchien werden als Pseudohierarchien bezeichnet und werden in einem eigenen Array gespeichert. Jedes Symbol, jedes Netz und jede Instanz erhält eine eigene ID. Um von der Bezeichnung auf die ID zu schließen, wird jeweils ein zweites Array eingeführt, welches als Schlüssel die Bezeichnung und als Wert die ID beinhaltet. So wird die Ausführungsgeschwindigkeit erheblich gesteigert.

Tabelle 3.2: Datenbasis

Array-Bezeichnung	Speicher
HIER_LEVEL	Hierarchie IDs zu jedem Level
COUNTER	Zähler für IDs, etc.
SYMBOL/ SYMBOLID	Symbolnamen/ID, Symbol-ID, Zelltyp, Pin-Bezeichnungen, Pin-Richtungen

Table 3.2 continued from previous page

Array-Bezeichnung	Speicher
NET/ NETID	Netznamen/ID, Sink-Pins, Source-Pins, Farbe/Clock, hierarchischer Pfad, Netzgruppe
INST/ INSTID	Instanzname/ID, Gattertyp, Symbol, Farbe/Clock, hierarchischer Pfad, Parent/Children, Pseudohierarchie Typ (Zelle/Hierarchie), Inhalt, angeschlossene Netze, verbundene Pins, Hierarchie, Exception-Pins
FLAGS	Flags
PORT	Portbezeichnung, Richtung
PSHIER/ PSHIERID	Bezeichnung/Pseudo-ID, Inhalt, Hierarchietyp, Eingangsnetz, hierarchischer Pfad, Pseudo-Unterhierarchie, Pseudo-Mutterhierarchie

Die in den NLV-Dateien enthaltenen API-Kommandos werden auf Tcl-Prozeduren abgebildet, sodass die einzelnen Zeilen beim Einlesen der Dateien einfach ausgeführt werden können. In dem folgenden Listing ist die Initialisierung solcher Prozeduren und das Einlesen der Dateien beschrieben. Die Abbildung der Prozeduren kann auch verkettet werden, was ebenfalls beispielhaft im Listing zu sehen ist. Es wird das API-Kommando "`load symbol ...`" abgebildet. Da mit dem zweite Argument des `load`-Kommandos mehrere unterschiedliche Objekte erzeugt werden können, wird das Argument ebenfalls auf mehrere Prozeduren abgebildet. Beispielsweise wird das Kommando "`load inst ...`" zur Erzeugung von Instanzen und "`load symbol ...`" zur Erzeugung von Symbolen verwendet. In Folge dessen werden die Prozeduren `inst` und `symbol` initialisiert. Innerhalb dieser Prozeduren werden anschließend die entsprechenden Datensätze angelegt.

```
1  proc load args { ;#Initialisierung des API-Tcl-Kommandos
2  eval $args
3  }
4
5  proc symbol {name view type args} { ;#Initialisierung des API-Tcl-Kommandos
6  ...
7  }
8
9  set fin [open ../Clocktree.nlv] ;#oeffne Datei
10 while {![eof $fin]} { ;#pares ueber die Zeilen
11 gets $fin line ;#erfasse Zeile
12 eval $line ;#fuehre das API-Kommando aus
```

```
13 }
14 close $fin ;#schließe die Datei
```

Quelltext 3.3: einlesen der NLV-Dateien

3.1.2 Prozeduren zur Datenerfassung

In diesem Abschnitt wird die Erfassung der Daten erläutert. Dazu werden Algorithmen verwendet, die ausgeführt werden, wenn ein auf Tcl abgebildetes API-Kommando als Prozedur aufgerufen wird. Für die Generierung der Symbole, Instanzen und Netze wird jedes mal ein Datensatz angelegt, welcher bei der Initialisierung aus der Bezeichnung und der ID besteht. Zum Setzen der IDs wird bei jeder Initialisierung ein passendes Element des Arrays `COUNTER` inkrementiert, beispielsweise "`incr INST(inst_id_ctr)`" für die IDs der Instanzen. Dessen Wert entspricht dann der neuen ID. In Abbildung 3.3 sind die beiden Prozeduren zum Einlesen der Symbol- und Instanzdaten schematisch dargestellt. Zu Beginn werden die Datensätze initialisiert und anschließend objektspezifische Daten ermittelt. Zuerst werden die Symbole erfasst. Die gewonnenen Symbolinformationen werden auch beim Auswerten der Instanzen verwendet. So können beispielsweise die Gattertypen der Instanzen festgelegt werden. Dies geschieht durch Auswertung des Argumentes `stype`. Dieses Argument enthält beispielsweise das Schlüsselwort "AND" und stellt ein Und-Gatter dar. Beim Einlesen der Instanzen wird der hierarchische Pfad zu den einzelnen Instanzen ermittelt. Dieser Pfad besagt, welche Hierarchien aufgefaltet werden müssen, um zu der entsprechenden Instanz zu gelangen und wird für eine Funktion verwendet, bei der es darum geht, Hierarchien zu separieren. Diese Funktion wird jedoch nicht im Rahmen dieser Arbeit behandelt.

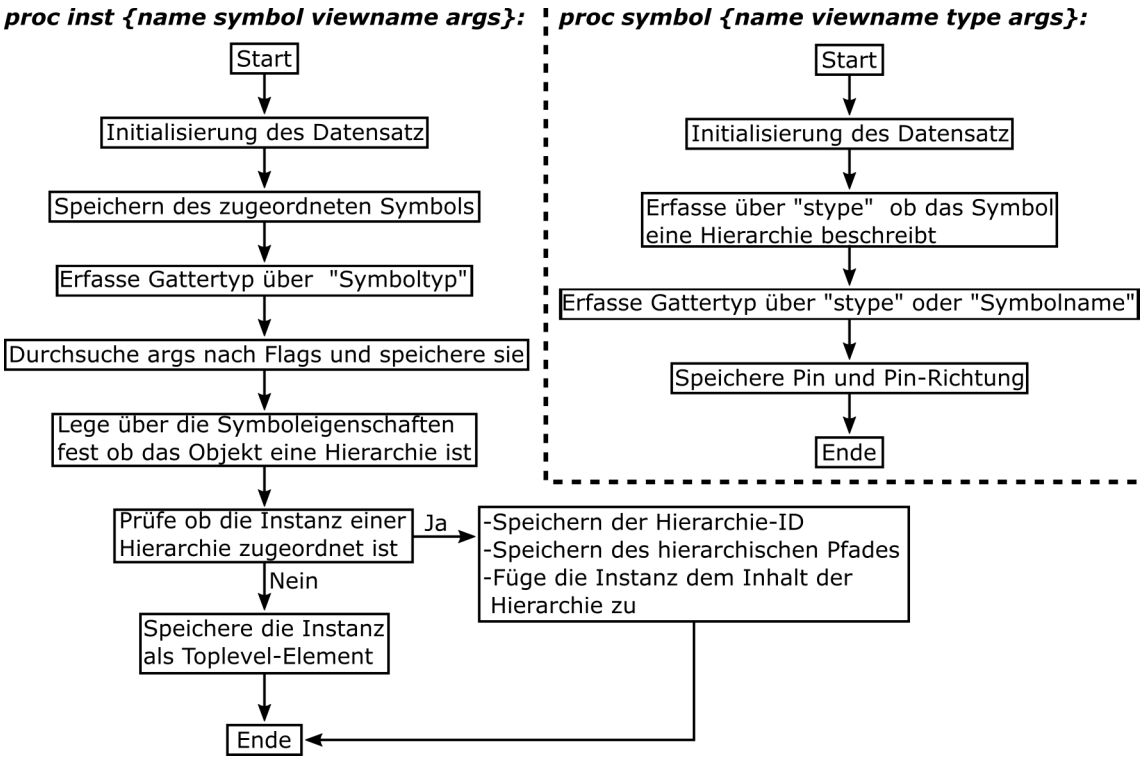


Abbildung 3.3: Erhebung der Symbol- und Instanzdaten

Weitere Prozeduren, welche beispielsweise die Ports in der Netzliste erfassen, haben ein ähnliches Schema. Auch die Erhebung der Netzdaten und die daraus hervorgehenden Informationen werden ähnlich ermittelt. Bei der Erfassung der Netze werden auch wichtige Informationen über die Struktur der Clocktrees ermittelt. Darauf wird in Kapitel 4 eingegangen, da diese Struktur für die Äquivalenzprüfung verwendet wird.

3.2 Qt-Anwendung

Die Qt-Anwendung basiert auf der NLview Demo-Applikation. Diese verfügt über einige GUI-Elemente die nicht im Clocktree-Analyser verwendet werden. Durch den Umbau der Demo-Anwendung wurden einige dieser Elemente entfernt und andere wurden ergänzt. Abbildung 3.4 zeigt die fertige Applikation, wie sie am Ausgangspunkt dieser Masterarbeit vorliegt. Die Menüleiste verfügt zusätzlich über die Menüs *Window* und *Clocktrees*. Unter *Window* kann die Minimap ein- und ausgeblendet werden. Diese ist auch Bestandteil der Demo-Applikation, der Menüeintrag, um diese aufzurufen, ist in der Demo aber unter dem Reiter *Options* zu finden. Über *Clocktrees* kann der anzuzeigende Clocktree ausgewählt werden. Rechts daneben befindet sich eine Statusanzeige. Diese ist grün, wenn keine Hintergrundprozesse ausgeführt werden und rot, wenn der Clocktree-Analyser arbeitet und keine Benutzerinteraktion möglich ist. Rechts in der Leiste werden Modul- und Viewname des aktuell dargestellten Moduls angezeigt. In der API-

Kommandozeile können über das Schlüsselwort "tcl" Tcl-Kommandos eingegeben werden. Die Ergebnisse werden in dem darunter liegenden Fenster ausgegeben.

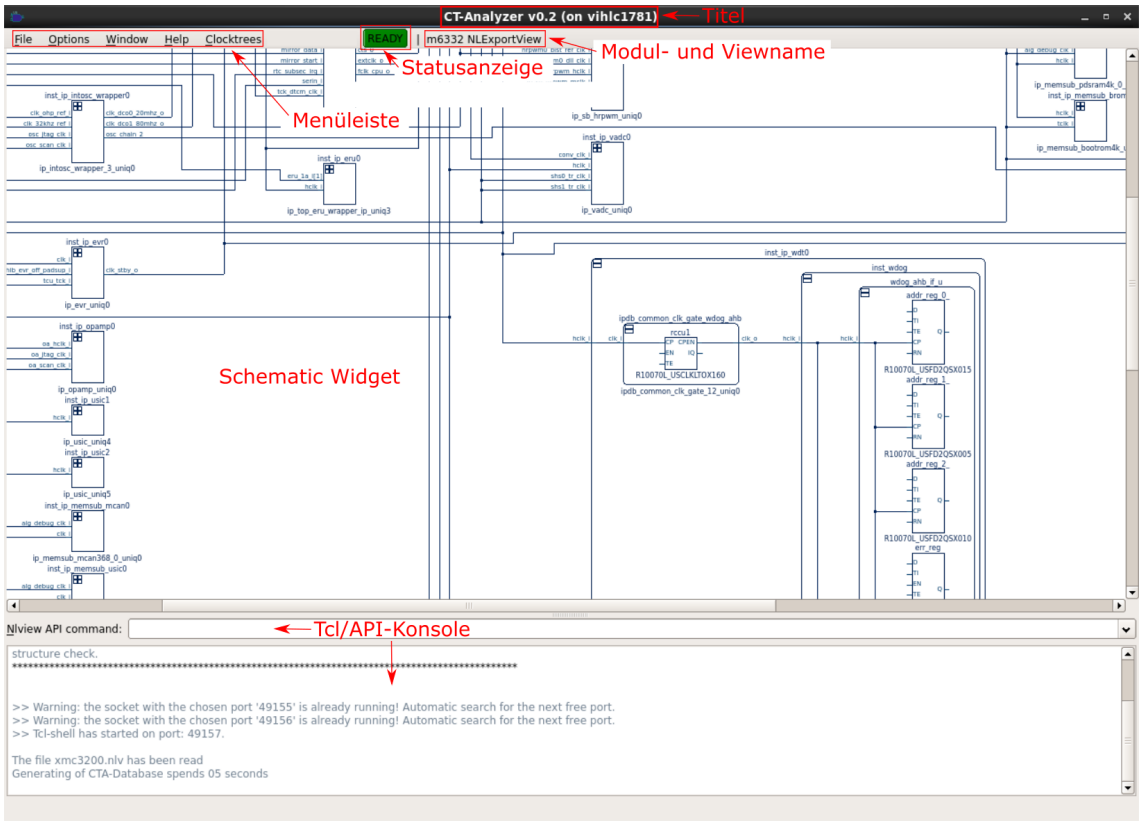
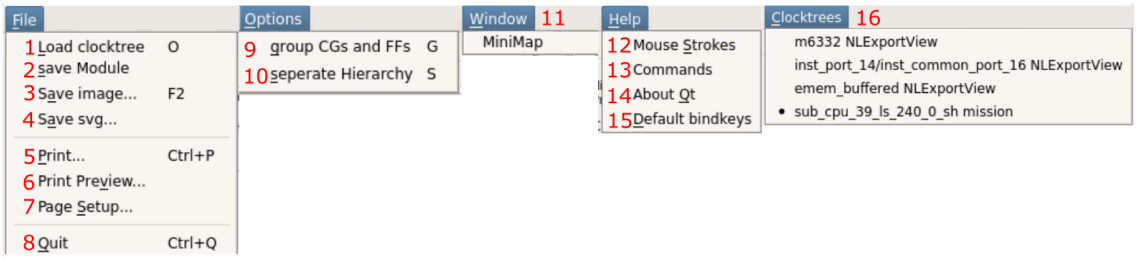


Abbildung 3.4: GUI-Aufbau des Clocktree-Analysers

In Abbildung 3.5 sind die Menüeinträge dargestellt. Die Einträge sind nummeriert und die Erklärung der Funktion ist der jeweiligen Nummer zugeordnet und befindet sich im unteren Abschnitt der Abbildung.



1: Laden einer Clocktree-Datei im nlv-Format
2: Speichern des aktuell angezeigten Moduls im nlv-Format
3: Speichern des aktuell angezeigten Moduls in Bildformaten wie JPG oder PNG
4: Speichern des aktuell angezeigten Moduls im SVG-Format
5-6: Drucken der aktuellen Widget anzeige
7: Festlegen des Seiten-Layouts, der zu druckenden Seite
8: Beenden des Programmes
9: Gruppieren der Flipflops und Clockgates Pseudo-hierarchien
10: Separieren ausgewählter Hierarchien
11: Anzeigen und ausblenden der Minimap
12-14: Anzeige von Anwendunginformationen
15: Einbinden von Standard-Mausbindings
16: Auswahl des anzuzeigenden Clocktrees

Abbildung 3.5: Menüleiste vom Clocktree-Analyser

Die Tcl-Anwendung wird in dem Konstruktor der Klasse Demo gestartet. Dazu wird ein Objekt der Klasse QProcess mit der Bezeichnung `tcl_server` erzeugt. Dieses Objekt startet eine Tcl-Shell, in der die nötigen Tcl-Dateien eingebunden werden. Das folgende Listing zeigt die Konfiguration und den Start der Shell im Konstruktor. Dabei wird ein Signal der Klasse QProcess mit dem Slot `readyReadStandardOutput()` getriggert, sobald Daten über einen der Standard-Datenströme verfügbar sind.

```
1  //erzeuge QProcess Objekt:
2  tcl_server =  new QProcess(this);
3  //Verbinde QProcess Signal mit eigener Funktion
4  connect(tcl_server, SIGNAL(readyReadStandardOutput()),
5  this, SLOT(readyReadStandardOutput()));
6  //führe die Datenströe zusammen:
7  tcl_server->setProcessChannelMode(QProcess::MergedChannels);
8  //starten der Tcl-Shell und sourcen der ersten Tcl-Datei
9  tcl_server->start("tclsh8.5",QStringList()<< nlv_home_dir.filePath("CtaMain.tcl"));
10 //warte bis der Server gestartet ist
11 tcl_server->waitForStarted(5000);
```

Quelltext 3.4: Konfiguration der Tcl-Shell in Qt

Innerhalb von `readyReadStandardOutput()` wird ausgewertet, ob die ankommenden Daten eine spezifische CTA-Systemnachricht enthalten. Dies ist der Fall, wenn der String "System_message" übertragen wurde. Wenn die Nachricht außerdem das Schlüsselwort "port" enthält, dann ist in diesem transferierten Datensatz die Port-Nummer des TCP/IP-Servers enthalten, der in der Tcl-Applikation erzeugt wird. Dieser wird in der Variablen `port` gespeichert. Über die interne IP-Adresse "127.0.0.1" und der Port-Nummer können beide Applikationen über eine TCP/IP-Verbindung Daten austauschen. Für einen solchen Datenaustausch muss jedes mal eine

neue Verbindung aufgebaut werden. Im nachfolgenden Listing ist die Initialisierung der Memberfunction `writeToTcl()` zu sehen. In dieser wird die Funktion `Connect()` aufgerufen, welche die Verbindung zum Tcl-Server herstellt. `writeToTcl()` wird ein String übergeben, der an die Tcl-Applikation übertragen wird. Daraufhin wird auf eine Antwort der Tcl-Anwendung gewartet. Diese Antwort ist anschließend der Rückgabewert der Funktion.

```
1 QString Demo::writeToTcl(QString message){
2 Connect();
3 tcl_socket->write(message.toUtf8());
4 tcl_socket->waitForBytesWritten(1000);
5 tcl_socket->waitForReadyRead(1000);
6 return tcl_socket->readAll().trimmed();
7 }
```

Quelltext 3.5: Initialisierung von writeToTcl()

Auch der Destruktor der Klasse Demo enthält neue Elemente. So ist nun die Zeile `writeToTcl("exit")` enthalten, um das CTA-Kommando "exit" an die Tcl-Anwendung, die durch die Initialisierung der Variablen `forever` beendet wird, zu transferieren. Der Destruktor wird beim der Beendigung der Qt-Applikation aufgerufen. Abbildung 3.6 zeigt die Änderungen in der Klasse Demo und erläutert kurz die Funktion der neuen Memberfunctions, Slots und Objekte.

Demo Anpassung	**Funktionalität**
writeToTcl()	Verbindet Qt und Tcl via TCP/IP, sendet einen String an Tcl und wartet bis die Verbindung wieder frei ist. Rückgabewert ist der Empfangene Antwortstring von Tcl.
Connect()	Verbindet Qt mit Tcl über die interne IP und dem gegebenen Port aus der Variablen *port*.
update_CT_title()	Setzt das Label *CT_title* auf den aktuellen Modulnamen.
slots:	
CTA_progressNotify()	Anzeigen des Status vom CTA durch die *ProgressBar* und den Button *busy_label*.
ct_default_bindkeys()	Einlesen einer NLV-Datei in Qt und Tcl
readct()	Komprimieren des CTs durch gruppieren der CGs und FFs.
group_cgff()	Separieren einer selektierten Hierarchie.
seperate_hier()	Anzeigen eines anderen Clocktrees über den entsprechenden Menüeintrag.
sel_module()	Empfangen von Daten über die Standard-Datenströme.
readyReadStandardOutput()	Speicherung des aktuell angezeigten Schematics
save_nlv()	
Objekte:	
QMenu* ctviewMenu	Auswahlmenü der einzelnen Clocktrees.
QMenu* windowMenu	Menü für das Anzeigen der Minimap.
QLabel* CT_title	Neben dem Menü angezeigte CT-Bezeichnung.
QPushButton* busy_label	Button zu Statusanzeige neben der Menüleiste.
QAction* actual_action	Wird zur Auswahl mehrerer Clocktrees benötigt.
QActionGroup* readfile_grp	Wird zur Auswahl mehrerer Clocktrees benötigt.
QTcpSocket* tcl_socket	Socket Element für die TCP/IP Verbindung zu Tcl.
QProcess* tcl_server	Enthält die Tcl-Shell.
quint16 port	Enthält die verwendete Port-Nummer der TCP/IP-Verbindung

Abbildung 3.6: Anpassungen in der Klasse Demo

Die Memberfunction `createMenus()` enthält neuen Code. Dort sind Menüeinträge für die Gruppierung von Clockgates und Flipflops sowie die Separation von Hierarchien ergänzt. Zudem sind Einträge zum Einlesen neuer Clocktrees und das Setzen von Standard-Bindkeys enthalten. Die Einträge werden als QActions in das Menü eingebunden. Außerdem wird das neue Menü für die Auswahl der Clocktrees durch den im unteren Listing dargestellten Code erzeugt.

```
1 optionsMenu->addAction(tr("&group CGs and FFs"), this, SLOT(group_cgff()));
2 optionsMenu->addAction(tr("&seperate Hierarchy"), this, SLOT(seperate_hier()));
3 fileMenu->addAction(tr("&Load clocktree"), this, SLOT(readct()));
4 helpMenu->addAction(tr("&Default bindkeys"), this, SLOT(ct_default_bindkeys()));
5 ctviewMenu = menuBar()->addMenu(tr("&Clocktrees"));
```

Quelltext 3.6: Erzeugen der neuen Menüeinträge

Die Auswahl der Clocktrees erfolgt über Menüeinträge, die als QActionGroup mit der Bezeichnung `readfile_grp` unter dem Reiter *Clocktrees* hinterlegt sind. Auf Grund der Eigenschaften der QActionGroup-Klasse kann immer nur einer dieser Menüeinträge aktiv sein. In der GUI wird der aktive Menüeintrag durch einen Haken gekennzeichnet. Wird im Menü ein anderer Eintrag

ausgewählt, dann wird dieser zur aktiven Action in der Gruppe. Jede der neuen Actions wird mit dem Slot `sel_module()` verbunden und erhält als Bezeichnung den Namen des damit verbundenen Clocktrees. Innerhalb von `sel_module()` wird die aktive Action von `readfile_grp` und die damit verbundene Bezeichnung ermittelt. Über diese Action kann anschließend der ausgewählte Clocktree im Schematic-Widget angezeigt werden. Die Erzeugung der Action findet beim Einlesen neuer Clocktrees statt. Im folgenden Listing befindet sich der zugehörige Quellcode. Die Variable `newActionName` enthält Modul- und Viewnamen des Clocktrees. Mit Hilfe der Variablen `actual_action` findet die Konfiguration der einzelnen Actions statt. Es ist auch der Slot `sel_module()` aufgeführt, an dessen Ende die Funktion `update_CT_title()` aufgerufen wird. Diese übergibt den Clocktree-Namen an das Label `CT_title`, welches den neuen Namen in der Menüleiste anzeigt.

```
1  //Anlegen der neuen Action:
2  actual_action = ctviewMenu->addAction(newActionName.toUtf8(), SLOT(sel_module()));
3  //Konfiguration der neuen Action:
4  readfile_grp -> addAction(actual_action);
5  actual_action->setCheckable(true);
6  actual_action->setChecked(true);
7
8  void Demo::sel_module(){
9  ...
10 //Erfasse Namen der aktiven Action (Modul- und Viewname)
11 QString CTName = readfile_grp->checkedAction()->text();
12 //Öffne den Clocktree
13 nlview->command(&ok, "module", "open", CTName.toUtf8());
14 update_CT_title();
15 }
```

Quelltext 3.7: Auswahl zwischen mehreren Clocktrees

Das Einlesen der Clocktrees basiert auf der Memberfunction `openNetList()`. Diese öffnet ein Fenster der Klasse QFileDialog. Darüber kann eine Datei ausgewählt werden, die anschließend über das API-Kommando "source" in NLView eingelesen und dargestellt wird. Abbildung 3.7 zeigt das neue Auswahlfenster, in dem einige NLV-Dateien zu sehen sind. Mit der Memberfunction `getOpenFileName()` kann der Pfad und der Name der geöffneten Datei ermittelt werden. Der Pfad und das CTA-Kommando "create_namespace" werden anschließend an die Tcl-Applikation übertragen, wo der neue Namespace erzeugt wird. Am Ende wird der neue Menüeintrag, wie oben erläutert, erzeugt.

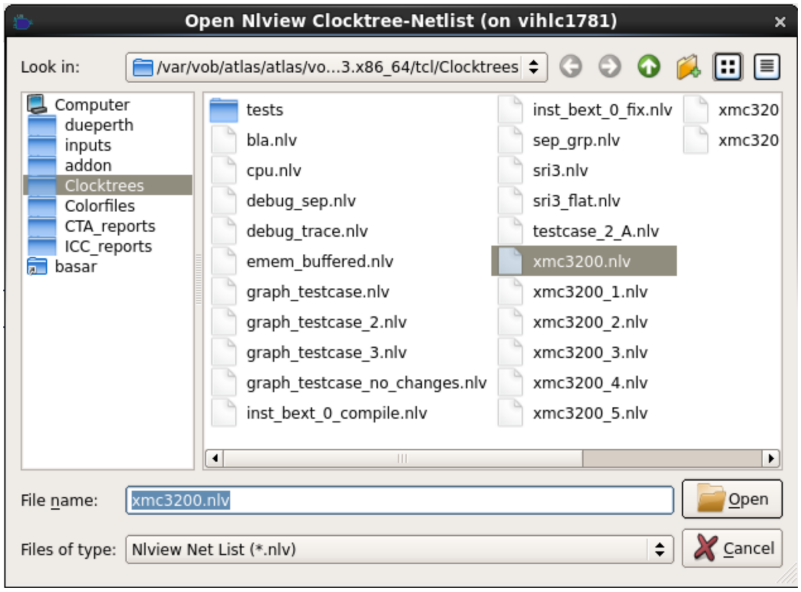


Abbildung 3.7: QFileDialog-Fenster zum Einlesen von NLV-Dateien

Auch die Speicherung der NLV-Dateien erfolgt über ein QFileDialog-Widget. Der in diesem Widget ausgewählte Pfad und der Dateiname werden an die Variable `choice` übergeben. Daraufhin wird geprüft, ob die Dateiendung ".nlv" in der Variablen erhalten ist. Falls nicht wird diese automatisch ergänzt. Anschließend wird über das API-Kommando `save` mit der Option `-curmodule` und der Angabe des Pfades und des gewählten Dateinamens das aktuelle Modul gespeichert. Im folgenden Listing ist ein Ausschnitt des erforderlichen Codes zu sehen.

```
1 QString choice = QFileDialog::getSaveFileName(this,"Save schematic nlv");
2
3 if( !choice.isNull() ) {
4 if( !choice.right(4).contains(".nlv")) {
5 choice.append(".nlv");
6 }
7 }
8 nlview->command(&ok, "save", "-curmodule", choice.toUtf8());
```

Quelltext 3.8: Speichern des aktuellen Moduls

Der Slot `evalCommand()` wird in der Demo-Version verwendet um Eingaben in der Kommandozeile auszuwerten. Dazu wird der Text aus dem Eingabefenster an diesen Slot übergeben und mit Hilfe von If-Abfragen analysiert. Dies ist im folgenden Listing beispielhaft dargestellt. Auch die Verwendung der Kommandozeile als Tcl-Konsole basiert auf diesem Prinzip. Dazu wird das Schlüsselwort "tcl" gefiltert und der Text aus der Kommandozeile via TCP/IP an die Tcl-Konsole übertragen. Das Schlüsselwort "tcl", das ebenfalls transferiert wird, stellt auch ein CTA-Kommando dar. Dieses wird in der Tcl-Shell ausgewertet und alle übergebenen Argumente als

Liste an das Tcl-Kommando `catch` übergeben. Es wird dann versucht das Kommando auszuführen, ohne einen Absturz der Tcl-Shell zu verursachen. Eine mögliche Fehlerausgabe wird an ein zweites Argument übergeben und als Variable gespeichert. Nachdem das Kommando ausgeführt wurde, wird die mögliche Fehlermeldung über den Standard-Datenstrom *stdout* ausgegeben. Auch die ausgeführten Befehle in dem Argument des `catch` Kommandos werden über diesen Ausgang ausgegeben. Dadurch können die Ausgaben direkt in der Qt-Applikation empfangen und in der GUI-Konsole angezeigt werden.

```
1 bool Demo::evalCommand(const QString& cmdline) {
2    if (cmdline.trimmed() == "exit") {
3       qApp->quit();
4       return true;
5    }
6    ...
7 }
```

Quelltext 3.9: Auswertung der Kommandozeile

4 Implementierung des Comparetools

Zur Feststellung der Äquivalenz zweier Clocktrees wird ein neues *Widget* eingefügt. Über dieses Widget können die Clocktrees ausgewählt und der Äquivalenz-Algorithmus gestartet werden. Der Algorithmus ist in der Tcl Applikation implementiert und basiert auf der internen Tcl-Datenbasis. Dazu wird die Clocktree-Struktur in der Datenbasis abgelegt, die bereits beim Einlesen der NLV-Dateien ermittelt wird. Die Ergebnisse der Analyse werden im Schematic farbig dargestellt. Instanzen, die in einem der beiden Clocktrees überzählig sind, werden farbig gekennzeichnet. Zudem werden die betroffenen Instanzen in einem Ausgabefenster aufgelistet.

4.1 Ausbau der Benutzeroberfläche

Die für den Anwender sichtbare Teil der Bedienoberfläche des Comparetools basiert auf einem *QDockWidget*, welches in Abbildung 4.1 dargestellt ist. Der Vorteil bei der Verwendung dieses Widgets liegt darin, dass das Widget bei Bedarf geschlossen und wieder geöffnet werden kann. Im oberen Bereich des Fensters befinden sich zwei Auswahlleisten, mit denen sich die zu vergleichenden Clocktrees auswählen lassen. Darunter ist ein Button platziert, der den Vergleich der Clocktrees startet und das CTA-Kommandos "compare" an die Tcl-Applikation transferiert. Rechts daneben befindet sich ein Auswahlkasten, über den festgelegt werden kann, ob eine detaillierte Analyse der Clocktrees oder eine schnelle Analyse stattfinden soll, bei der lediglich eine Aussage über die Isomorphie der beiden Clocktrees getroffen wird. Sobald dieser Auswahlkasten gesetzt ist, erscheinen unterhalb des Start-Buttons zwei weitere Kontrollkästchen. Mithilfe dieser Auswahlflächen kann bestimmt werden, wie die detaillierte Analyse ausgeführt werden soll. Dabei kann gewählt werden, ob die reine Clocktree-Struktur analysiert werden soll oder ob die Analyse auf dem Vergleich aller Instanz-Namen basiert. Während des strukturellen Vergleichs findet auch eine Priorisierung der Namen statt. Dabei werden Strukturpfade, deren Instanzbezeichnungen gleich sind, gegenüber der Strukturähnlichkeit bevorzugt. Weiteres dazu ist in Abschnitt 4.5 erläutert. Das Ergebnis der Detailanalyse ist eine Liste der Instanzen, die in einem der beiden Clocktrees überzählig sind. Mit dem rechten der beiden unteren Buttons kann diese Liste in der Ergebnisanzeige ausgegeben werden. Während der Detailanalyse können Strukturen auftreten, die nicht für einen Vergleich erfassbar sind. Diese nicht analysierbaren Pfade werden mithilfe des linken Buttons in der Ergebnisanzeige dargestellt.

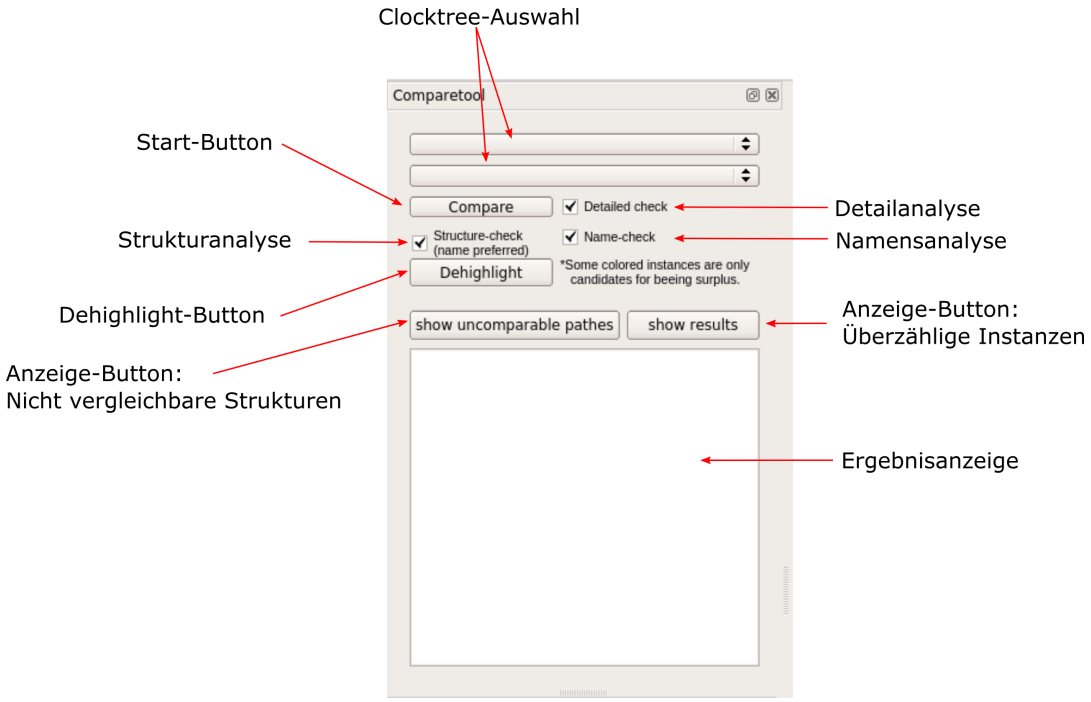


Abbildung 4.1: Benutzeroberfläche des Comparetools

Während der detaillierten Strukturanalyse des Clocktree-Vergleichs tritt ein grundsätzliches Problem auf, das auf Grundlage der Graphentheorie nicht gelöst werden kann. Das Problem besteht darin, bei unterschiedlichen Graphen-Strukturen zu ermitteln, welcher Knoten explizit den Unterschied in den Strukturen verursacht. In Abbildung 4.2 sind zwei Clocktree-Strukturen dargestellt. In Struktur a befindet sich ein Flipflop weniger als in Struktur B. Über die reine Strukturanalyse ist es nicht möglich den überzähligen Knoten aus Struktur B zu ermitteln, da jedes Flipflop dafür infrage kommen kann. Erst durch die Überprüfung der Bezeichnungen (FF_1, FF_2, usw.) kann FF_3 als überschüssig identifiziert werden. Dieses Kriterium reicht jedoch nicht für eine eindeutige Aussage aus, da die Namen aller drei Flipflops unterschiedlich sein könnten. Ein Info-Text, der in Abbildung 4.1 neben dem Dehighlight-Button zu sehen ist, weist auf diesen Umstand hin. Der Algorithmus ermittelt in einem solchen Fall alle infrage kommenden Kandidaten.

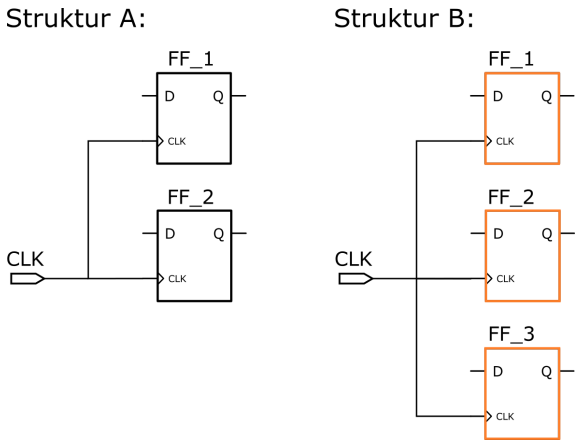


Abbildung 4.2: Problematik des Strukturvergleichs

4.1.1 Implementierung der Benutzeroberfläche in Qt

Die Implementierung des Comparetools findet in der Qt-Anwendung statt. Dazu wird die Memberfunction `addCompDock()` der Klasse `Demo` initialisiert. Innerhalb dieser Klasse wird ein Objekt `compdock` der Klasse `QDockWidget` erzeugt, welchem die Auswahllisten `ct_sel_A` und `ct_sel_B` hinzugefügt werden. Diese Listen sind vom Typ `QComboBox` und werden zur Auswahl der Clocktrees verwendet. Die Bezeichnungen der Clocktrees werden den Listen nach dem Einlesen der NLV-Dateien zugeordnet. Das folgende Listing zeigt diese Zuordnung. Die Variable `actModName` enthält dabei den Modul- und Viewnamen des eingelesenen Clocktrees.

```
1 Demo::readct() {
2    ...
3    ct_sel_A->addItem(actModName.toUtf8());
4    ct_sel_B->addItem(actModName.toUtf8());
5    ...
6 }
```

Quelltext 4.1: Übergabe der CT-Bezeichnungen zu den Auswahllisten

Auch die anderen Elemente, die in Abbildung 4.1 dargestellt sind, werden in der Funktion `addCompDock()` deklariert und dem Widget `compdock` zugeordnet. Daraufhin wird, wie im Listing 4.2 aufgeführt, das `QDockWidget` an das Mainwindow angekoppelt. Anschließend wird eine `QAction` erzeugt, mit der es möglich ist, dieses Widget zu schließen und wieder zu öffnen. Diese Action wird dem Menüeintrag Window zugeordnet und erhält die Bezeichnung "Comparetool".

```
1 //Integration des Comparetools in das Mainwindow:
2 addDockWidget(Qt::LeftDockWidgetArea, compdock);
3 //Erzeugung der Action zum Öffnen und Schließen des Comparetools:
4 QAction* compAct = compdock->toggleViewAction();
5 compAct->setStatusTip("Comparetool");
```

```
6 windowMenu->addAction(compAct);
```

Quelltext 4.2: Verknüpfungen zwischen Comparetool und Mainwindow

Wie im Listing 4.3 werden am Ende der Funktion Signale der GUI-Elemente mit speziellen Slots gekoppelt. Die Funktionsweise der Slots wird dabei im Folgenden vorgestellt.

```
1  //Detailanalyse Checkbox:
2  connect(det_comp_check,SIGNAL(stateChanged(int)), this, SLOT(detComp(int)));
3  //Startbutton
4  connect(compButton,SIGNAL(clicked()), this, SLOT(compare()));
5  //Anzeige-Button (nicht vergleichbare Strukturen):
6  connect(show_pathes_button,SIGNAL(clicked()), this, SLOT(showPathes()));
7  //Anzeige-Button (überzählige Instanzen):
8  connect(show_res_button,SIGNAL(clicked()), this, SLOT(showRes()));
9  //Dehighlight-Button:
10 connect(dehighlight_comp_button,SIGNAL(clicked()), this, SLOT(dehighlightComp()));
```

Quelltext 4.3: Verknüpfungen der Comparetool-Signale

Der folgende Quellcode zeigt den Slot `detComp(int)`, der mit dem Auswahlkasten für den detaillierten Vergleich der Clocktrees verknüpft ist. Beim Wechsel des Zustandes des Kontrollkastens wird dieser Slot getriggert. Dabei wird ein Integer übergeben, der den Status des Kastens ausdrückt. Wenn der Integer den Wert "0" besitzt, ist der Kasten nicht aktiv, während beim Wert "1" der Kasten aktiviert wurde. Innerhalb des Slots wird dieser Status überprüft. Beim Status "0" werden die Auswahlkästen für die Strukturanalyse und die Namensanalyse sowie der Infotext ausgeblendet. Andernfalls werden diese Elemente angezeigt.

```
1  det_comp(int state) {
2     if(state == 0) {
3        comp_infos->setText("");
4        struct_check-> setVisible(false);
5        name_check-> setVisible(false);
6     } else {
7        QString infoText = "*Some colored instances are only\n   candidates for
             beeing surplus.";
8        comp_infos->setText(infoText);
9        struct_check-> setVisible(true);
10       name_check-> setVisible(true);
11    }
12 }
```

Quelltext 4.4: Slot: detComp(int)

Im Slot `compare()` wird die Äquivalenzanalyse gestartet und die überzähligen Instanzen eingefärbt. Dieser Vorgang ist in Abbildung 4.3 dargestellt. Als erstes werden die in den Aus-

wahllisten selektiert Clocktrees erfasst. Daraufhin wird über die Auswertung der Checkbox `det_comp_check` überprüft, ob eine detaillierte Analyse stattfinden soll. Anschließend wird das CTA-Kommando zusammengestellt. Dieses besteht aus dem Schlüsselwort "compare" gefolgt von den Bezeichnungen der Clocktrees und der Information, ob die Analyse detailliert oder vereinfacht durchgeführt werden soll. Am Schluss des Kommandos sind die Werte der Auswahlkästchen für die Struktur- und die Namensanalyse angehängt. Im Falle einer schnellen Analyse werden diese auf "0" gesetzt. Anschließend wird das Kommando an die Tcl-Applikation übertragen. Bei der schnellen Analyse wird der Vorgang daraufhin beendet. Die Übertragung der Ergebnisse findet in dann durch die Standarddatenströme statt. Die Ergebnisse einer detaillierten Analyse stellen möglicherweise eine große Datenmenge da. Daher werden in diesem Fall die Daten über die TCP/IP-Verbindung transferiert. Während der Übertragung werden erst die Bezeichnung des Moduls und anschließend die Datensätze, welche die überzähligen Instanzen und Elemente enthalten, übertragen. Ein solcher Datensatz besteht aus einem Typ, dem Namen des Objektes und einer Farbe, in der das Objekt im Schematic eingefärbt werden soll. Der Typ kann entweder eine Instanz, ein Netz oder eine Konsolenausgabe beschreiben. Der Ausgabetyp lautet: "tool_log". Diese Zuordnung ist wichtig, da nicht alle Instanzen und Netze in der Ergebnisanzeige ausgegeben werden sollen. In der Qt-Anwendung fehlen die nötigen Informationen, um die signifikanten Elemente zu ermitteln. Diese Daten werden in einem Objekt mit der Bezeichnung "module_hash" des Typs QHash abgelegt. Dieser Datentyp ist ähnlich wie ein Tcl-Array. Auf die Inhalte kann über eine Schlüssel-Wert-Paar Beziehung zugegriffen werden. Als Schlüssel wird der Modulname, die Typzuordnung (z.B: inst oder tool_log) und eines der beiden Schlüsselwörter "comp_res" oder "uncomp_res" verwendet. Die Schlüsselwörter ermöglichen die Unterscheidung zwischen Elementen, die einem nicht vergleichbaren Pfad angehören und den Elementen, die überzählig sind. Die Werte des QHashes sind eine Liste von Instanzen und Netzen. Anschließend werden die Elemente im Schematic in zwei Farben koloriert. Eine Farbe beschreibt die Elemente, die überzählig sind und die zweite, ob das Element sich in einer nicht vergleichbaren Struktur befindet. Die Farben werden den Elementen über das NLV-Attribut @color zugewiesen. Der Code aus der Abbildung ist aus Gründen der Übersichtlichkeit nur an den Originalen Code angelehnt und nicht funktionsfähig.

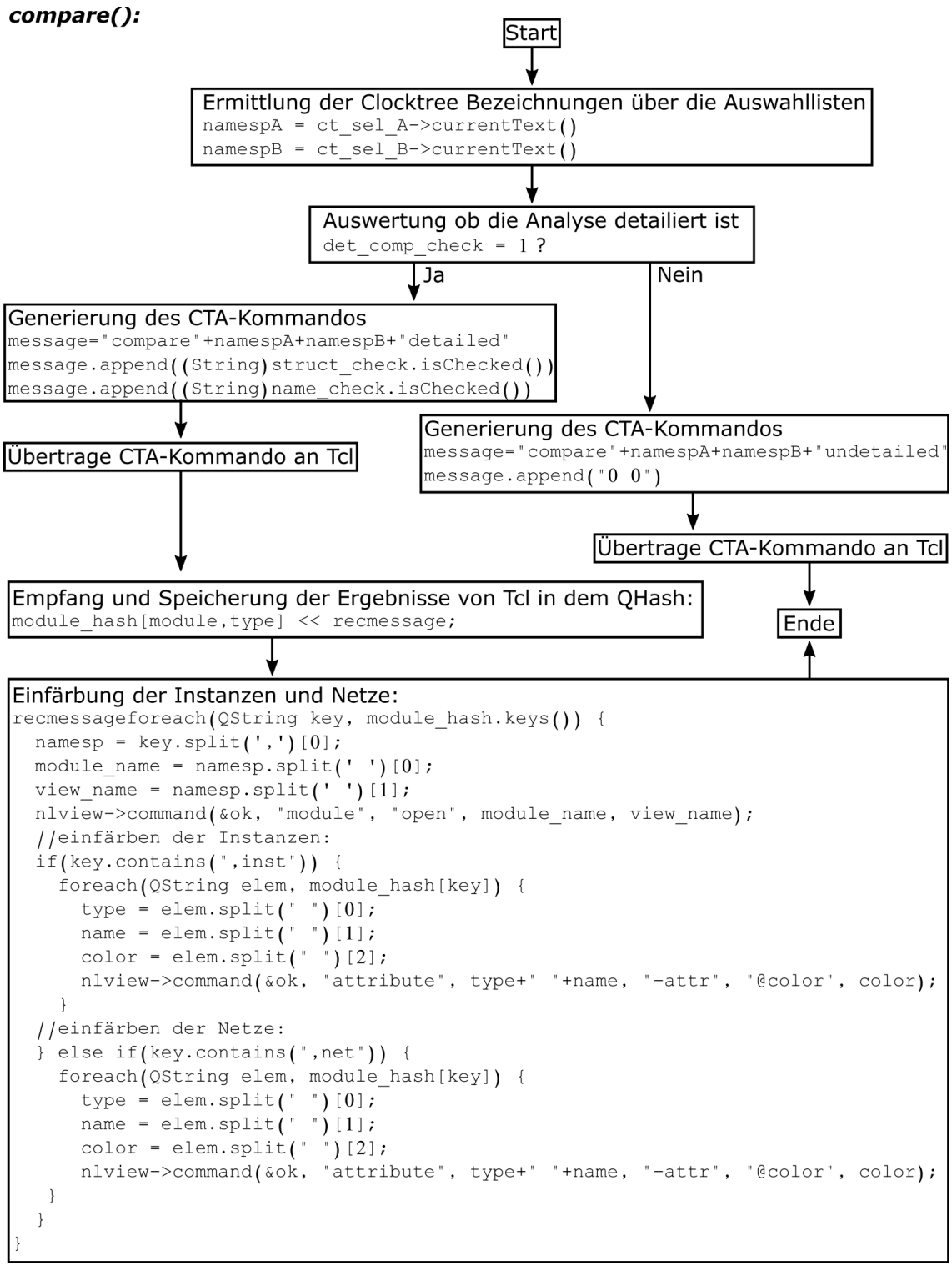


Abbildung 4.3: Schaubild des Slots `compare()`

Die Slots `showPathes()` und `showRes()` funktionieren ähnlich. Sie sind dazu da, um die

Ergebnisse und die Elemente, die sich innerhalb der nicht vergleichbaren Pfade befinden, in dem Ausgabefenster anzuzeigen. Dabei werden die Schlüssel des Objektes `module_hash` gefiltert. Die Kriterien des Filters sind je nach Slot das Schlüsselwort "tool_log" und eines der Wörter "comp_res" oder "comp_res".

Der Slot `dehighlightComp()` wird verwendet, um die Farben im Schematic wieder zu entfernen. Der notwendige Algorithmus ist ähnlich dem der zur Kolorierung verwendet wird. Die Löschung der Farbinformationen wird mithilfe des API-Kommandos "`attribute -remove type name -attr @color`" erreicht.

4.2 Erfassen der Clocktree-Struktur

In diesem Kapitel wird das Einpflegen der Clocktree-Struktur in die Datenbasis beschrieben. Die Struktur besteht aus den Clocktree-Elementen und deren Verbindungen (Netzen) untereinander. Der Clocktree ist dabei ausgeflacht, das heißt, dass in der Struktur keine hierarchischen Zellen berücksichtigt werden. Wie in Abbildung 4.4 dargestellt, sind in der einzulesenden NLV-Datei sind die Netze durch die Pins der Hierarchien getrennt. Netz_1 und Netz_2 sind strukturell miteinander verbunden, werden jedoch durch die Hierarchie in zwei Netze aufgetrennt. Diese strukturellen Verbindungen werden im weiteren Verlauf als Signale bezeichnet. Netz_1 und Netz_2 ergeben also ein Signal. Das gilt auch für Netz_3 und Netz_4. Darüber hinaus ist zu sehen, dass die Pins auf der Innenseite von hierarchischen Zellen als "hierPin" bezeichnet werden. Zudem ist zu erwähnen, dass die Pin-Richtung dieser Pins umgekehrt ist. Auch dies ist der Abbildung zu entnehmen.

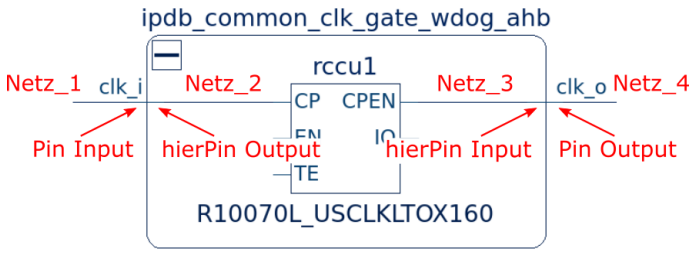


Abbildung 4.4: Hierarchie-Pins bei der Netzgruppen-Erfassung

Zur Erfassung der Struktur werden alle Netze, die zu einem Signal gehören in Netzgruppen zusammengefasst. Außerdem werden alle Instanzen, die mit dem jeweiligen Signal verbunden sind, diesen Gruppen zugeordnet. In Abbildung 4.5 ist ein Signal dargestellt. Die Instanzen, die über einen Ausgangs-Pin mit dem Signal verbunden sind, werden als Parents bezeichnet. Instanzen, die mit einem Eingangs-Pin angeschlossen sind, werden als Children bezeichnet. Das Signal in der Abbildung besteht aus den vier benannten Netzen und den Netzen, die sich in den zusammen-

gefalteten Hierarchien befinden und über Pins mit dem Signal verbunden sind. Ein Signal kann aus mehreren Eingangs- und Ausgangsnetzen bestehen. An Netz_4 sind Primitives ausschließlich mit ihren Eingangs-Pins verbunden. Der hierarchische Pin ist ein Eingang, daher kann Netz_4 als Ausgangsnetz deklariert werden. Netz_1 zeichnet sich als Eingangsnetz aus, da das verbundene Clockgate mit einem Ausgangs-Pin verbunden ist und der hierarchische Pin ein Ausgangs-Pin ist.

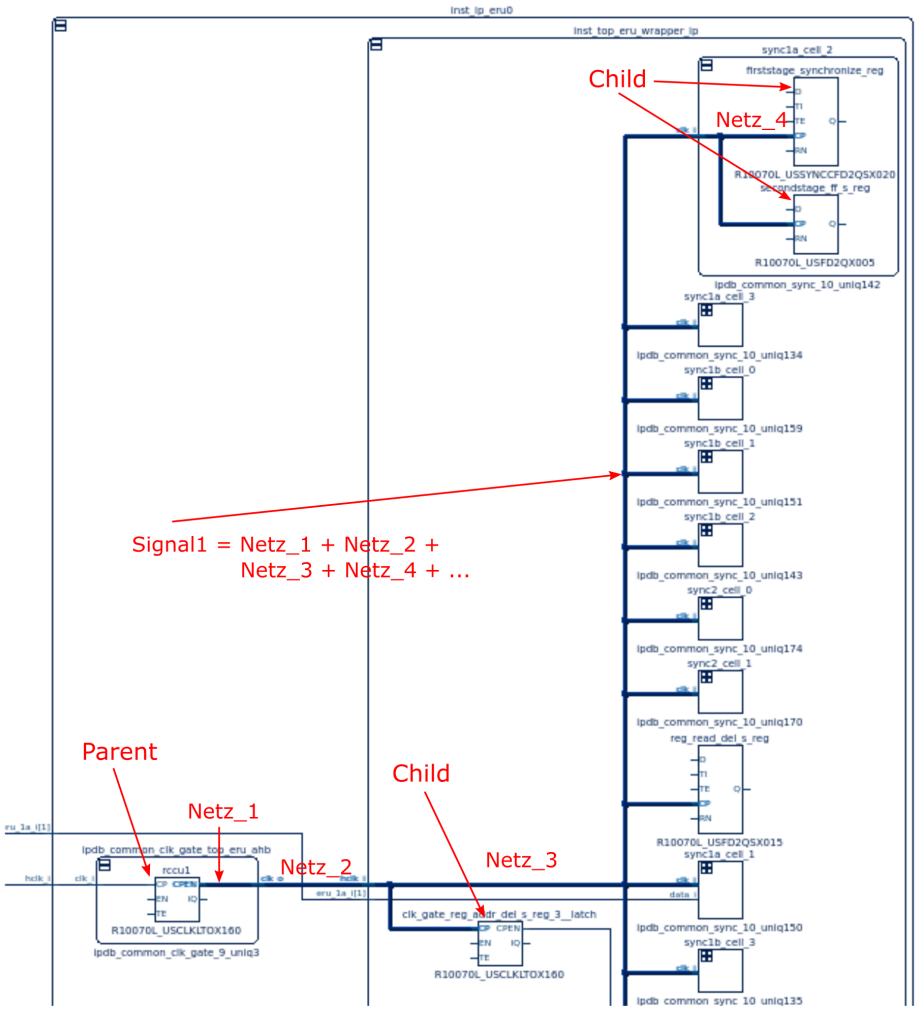


Abbildung 4.5: Zusammenstellung der Netzgruppen

4.2.1 Algorithmen zum Einpflegen der Clocktree-Struktur in die Tcl-Datenbasis

Das Einpflegen der Struktur findet in der Prozedur `net` statt. Diese ist auf das gleichnamige API-Kommando abgebildet und wird so bei der Erzeugung der Datenbasis ausgeführt. Abbildung 4.6 zeigt den Algorithmus, welcher in der Prozedur abläuft. Es ist zu sehen, dass nach dem Start der Netz-Datensatz initialisiert und dabei im entsprechenden Array abgelegt wird. Daraufhin

werden alle Attribute, die für die weitere Analyse nicht benötigt werden, entfernt, sodass nur noch die Liste der Verbindungen übrig bleibt. Daraufhin wird jedes Element dieser Liste in einer `foreach`-Schleife betrachtet. Ein Element besteht aus einem der beiden Schlüsselwörter "-pin" oder "-hierPin", dem Instanznamen und der Pinbezeichnung. Die verbundenen Pins werden im Datensatz der Instanz gespeichert. Über die Referenz des Symbols der aktuell betrachteten Instanz kann die Richtung des Pins erfasst werden. Dabei werden die "hierPins" und deren Richtungswechsel nicht berücksichtigt, da diese nicht extra bei der Symbolbeschreibung angegeben werden. So hat eine Hierarchie mit einem Eingang in der Symbolbeschreibung nur einen Eintrag für diesen Pin und dieser ist als "input" deklariert. Ist ein "hierPin" in der Liste vorhanden, so muss die Richtung getauscht werden. Dies ist in den Abbildungen 4.7 dargestellt. Im nächsten Schritt wird die Position des betrachteten Netzes in einer Netzgruppe erfasst. Es wird ermittelt, ob sich das Netz am Anfang, am Ende oder in der Mitte befindet. Außerdem werden dem Netz alle verbundenen Instanzen als Parents oder Children zugeordnet. Dieser Schritt wird im Folgenden genauer erläutert. Anschließend wird der nächste Zyklus der Schleife durchgeführt oder sie wird verlassen. Zum Schluss wird analysiert, ob das Netz durch Hierarchien getrennt und ob eine Netzgruppenzugehörigkeit gegeben ist. Wenn dies nicht der Fall ist, können allen mit dem Netz verbundenen Parents die Children und den Children die Parents zugeordnet werden. Diese Informationen werden im Array `INST` hinterlegt.

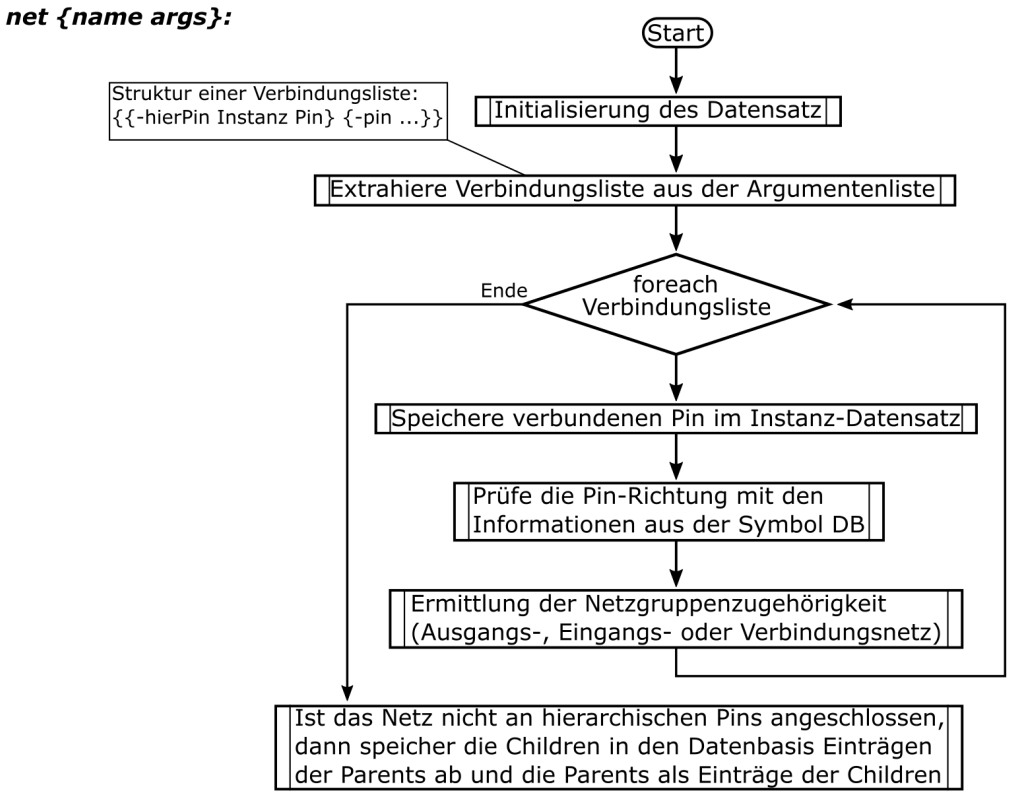


Abbildung 4.6: Erfassung der Netzeigenschaften

Die Abbildungen 4.7 beschreibt die Ermittlung der Netzgruppenzugehörigkeit des Netzes und die Initialisierung der Parents und Children. Diese setzt nach der Erfassung der Pin-Richtung an. Die orange eingefärbten Schritte zeigen den Ablauf für Eingangs-Pins und die grünen demonstrieren den Programmablauf für Ausgangs-Pins.

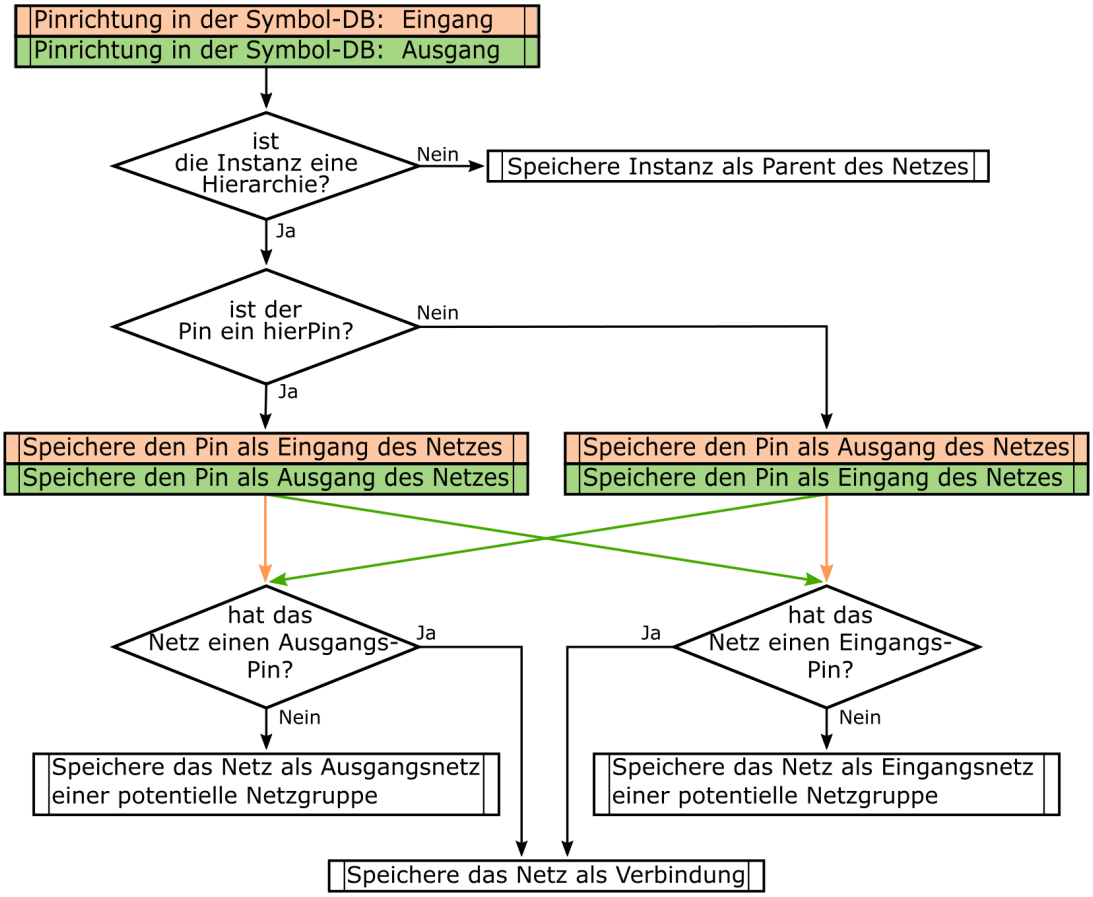


Abbildung 4.7: Ermitttlung der Netzgruppenzugehörigkeit für Ausgangs- und Eingangspins

Als letzter Schritt werden die Netzgruppen zusammengestellt. Dieser Vorgang wird in Abbildung 4.8 beschrieben, er wird in der Prozedur `get_design_structure`, nach dem Einlesen der NLV-Datei, ausgeführt. Dabei wird zuerst eine Liste aller Eingangsnetze zusammengestellt. Mithilfe einer `foreach`-Schleife wird jedes Element dieser Liste untersucht. Für jedes Eingangsnetz wird eine Netzgruppe generiert. Daraufhin wird eine temporäre Liste erzeugt. Diese beinhaltet zuerst nur das Startnetz, wird aber im weiteren Verlauf des Algorithmus mit allen Netzen, die zur gleichen Netzgruppe gehören wie das Startnetz, gefüllt und anschließend gelöscht. Daraufhin wird sie für die nächste Gruppe neu angelegt.

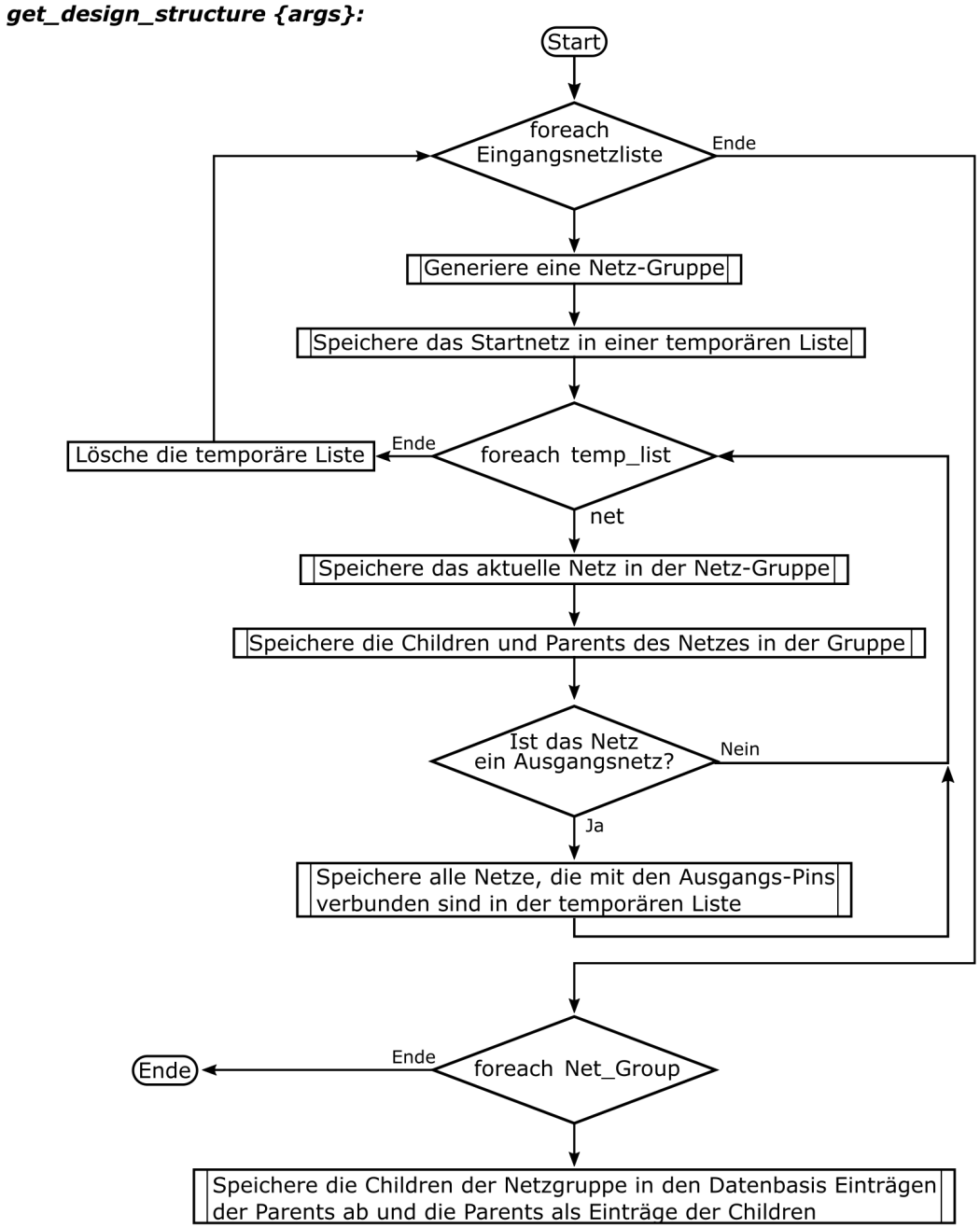


Abbildung 4.8: Auswertung der Netzgruppen

4.3 Extraktion der Graphen aus der Clocktree-Struktur

Die Graphen, die zur Überprüfung der Äquivalenz verwendet werden, werden aus der Clocktree-Struktur extrahiert. Dazu werden erst Instanzen herausgesucht, die als Wurzeln der Graphen dienen. Es werden also pro Clocktree mehrere Graphen erzeugt, die später mit denen eines anderen

Clocktrees verglichen werden. Abbildung 4.9 zeigt diese Extraktion. Dabei wird jede Instanz zu einem Knoten des Graphen. Sonderfälle, die über unterschiedliche Pins mit mehreren Parents verbunden sind, wie der Multiplexer in der Abbildung, werden zu mehreren Knoten. So kann aus dem Clocktree ein Baum generiert werden, bei dem jeder Knoten genau einen Parent hat. Dazu werden die Pins in einer Liste sortiert. Das erste Element in dieser Liste wird zu einem normalen Knoten transformiert. Alle anderen Pins in der Liste werden zu Pseudoknoten deklariert. Diese werden als Blätter in den Graphen eingepflegt. Die Namen der Knoten entsprechen den IDs der Instanzen, von denen sie abgeleitet sind. Die Pseudoknoten erhalten eine Pseudo-ID, die aus der Instanz-ID und einem inkrementierten Wert bestehen.

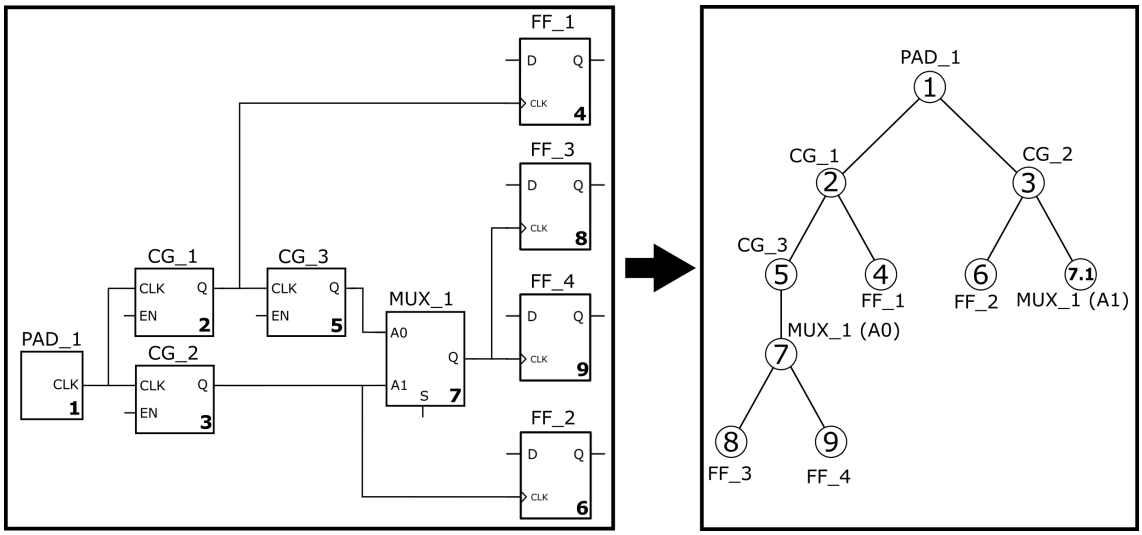


Abbildung 4.9: Extraktion eines Graphen

Mithilfe der Prozedur `get_roots` werden alle Instanzen ermittelt, welche eine Wurzel eines Graphen darstellen. Dazu wird überprüft, welche Instanzen keine vorgeschalteten Instanzen haben. Hierarchien sind generell als Wurzeln ausgeschlossen. In Abbildung 4.10 ist die Funktionalität der Prozedur schematisch dargestellt. Nachdem alle Wurzeln gefunden wurden wird die Prozedur `generate_graphs` ausgeführt. Diese Prozedur erzeugt die Tcl-Baum-Graphen anhand der ermittelten Wurzeln und speichert die Bezeichnung des Graphen in dem Array "TREES", welches im Namespace des betrachteten Clocktrees abgelegt ist. Über diese Bezeichnung kann, wie in Abschnitt 2.2.3 erläutert, auf die Prozeduren der Baum-Objekte zugegriffen werden.

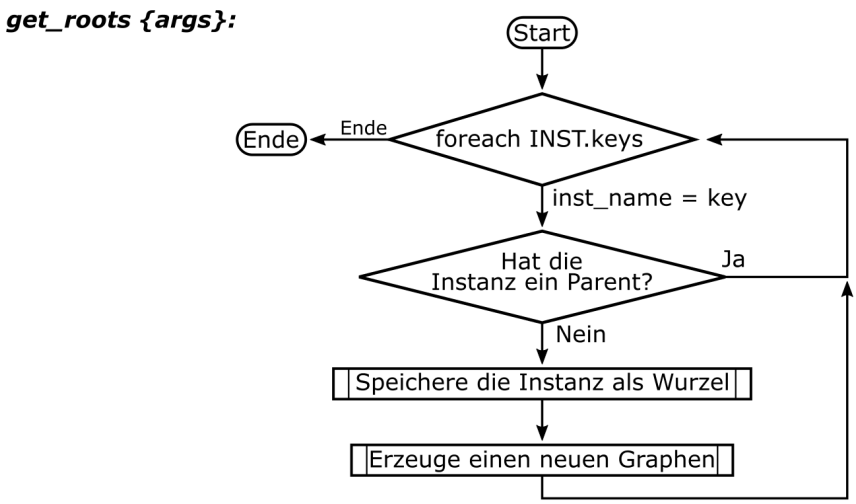


Abbildung 4.10: Ermittlung der Wurzeln

Das Prinzip der Graphenerzeugung beruht darauf, dass die Children eines Parents untersucht und gegebenenfalls als Knoten einem Graphen zugeordnet werden. Daraufhin werden die Children des aktuell untersuchten Childs einem Array als Wert hinzugefügt. Der zugehörige Schlüssel ist das aktuelle Child. Anschließend werden alle Elemente dieses Parent/Child-Arrays untersucht. Auf diese Weise kann der Graph aufgebaut werden. Das erste Element im Array ist die Wurzel. Während dieses Vorgangs ist darauf zu achten, ob eine Instanz bereits aufgerufen wurde. In diesem Fall ist diese Instanz über einen zweiten Pin im Clocktree verbunden und es wird ein Pseudoknoten generiert. Abbildung 4.11 zeigt den Algorithmus zur Erzeugung und Einpflegung der Knoten. Die Instanzen sind mit IDs versehen, die jeweils in der rechten unteren Ecke der Instanz zu sehen sind. Diese IDs sind auch die Bezeichnungen, die in den Knoten stehen. Zusätzlich sind die Knoten mit den entsprechenden Instanz-Bezeichnungen versehen. Ferner ist zu erwähnen, dass die Bezeichnungen der Instanzen als Attribute der Knoten gespeichert werden.

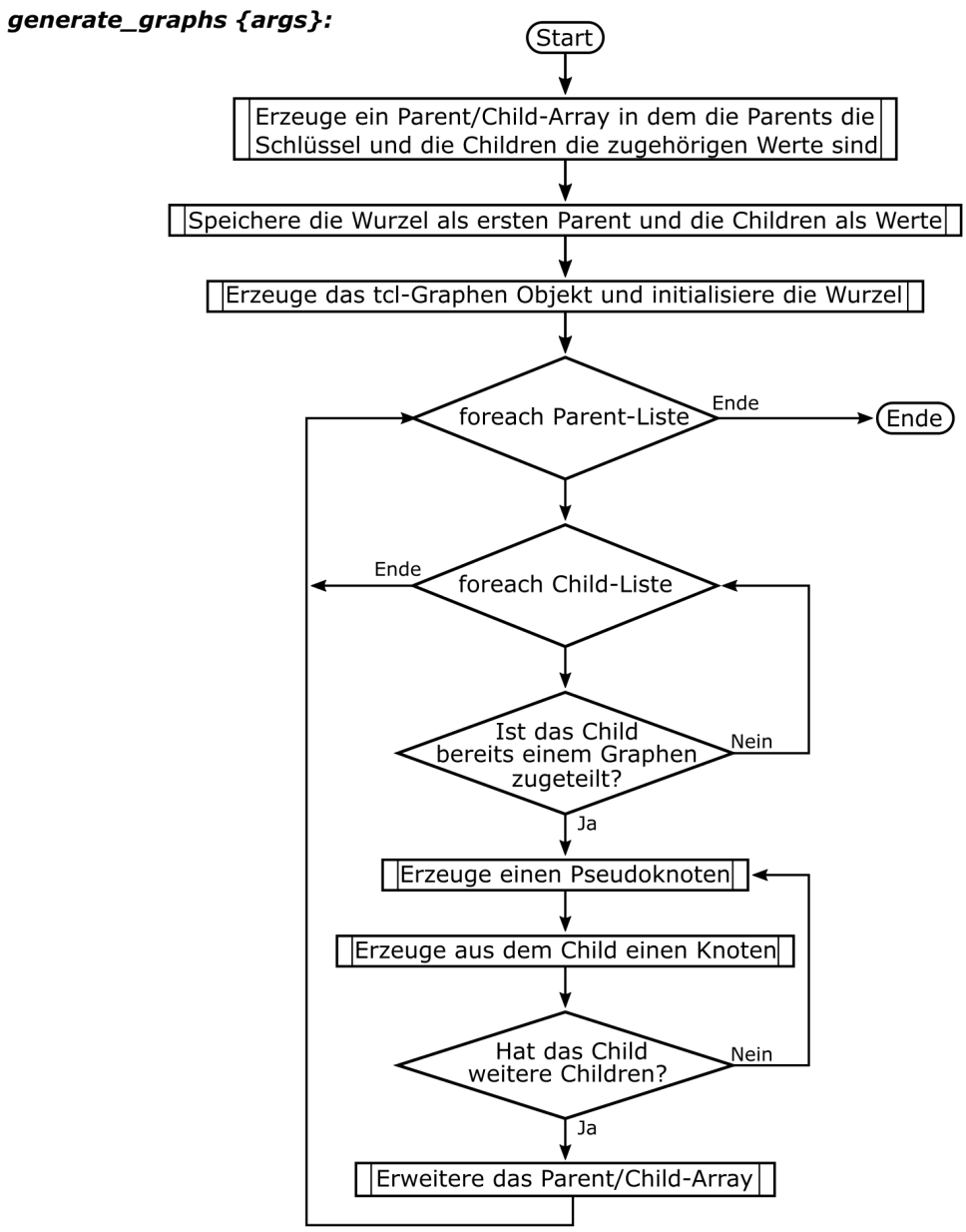


Abbildung 4.11: Einpflegen der Knoten in einen Graphen

4.4 Generierung der Strukturdatenbasis

Wie in Abschnitt 2.2.2 erläutert, wird die Isomorphie zwischen zwei Graphen auf der Grundlage von Strukturzahlen (bzw.Struktur-IDs) festgestellt, welche unabhängig vom Graphen erzeugt werden. Das heißt, dass die gleichen Strukturen in unterschiedlichen Graphen die selben Strukturzahlen besitzen. Um dies in der Tcl-Applikation umzusetzen, wird eine Datenbasis erzeugt, die nicht Teil der Clocktree-Namespaces ist und im globalen Namensraum abgelegt wird,

sodass aus jedem Namespace darauf zugegriffen werden kann. Die Erzeugung der Datenbasis erfolgt unmittelbar vor dem Äquivalenz-Algorithmus im Rahmen der Ausführung des CTA-Kommandos `compare`. Das Array, in dem die Strukturen abgelegt sind, hat die Bezeichnung "STRUCTURE_DB". Es erhält als Schlüssel die Struktur-ID und als Wert einen String, der die Strukturzahlen der Children enthält, die durch einen "." voneinander getrennt sind. Zudem wird der Instanztyp an das Ende des Strings angehängt. Das Referenzarray, mit dem vom Struktur-String auf die Struktur-ID geschlossen werden kann, ist als "STRUCTURE_DB_ID" bezeichnet. Bevor die Datenbasis erzeugt wird, wird geprüft, ob die Strukturen der ausgewählten Clocktrees bereits analysiert und in der Strukturdatenbasis gespeichert wurden.

Abbildung 4.12 zeigt das Schema der Prozedur `generate_structure_DB` zur Erzeugung der Strukturdatenbasis. Dabei wird als erstes das Array `TREES` in seine Schlüsselwörter segmentiert und mithilfe einer `foreach`-Schleife analysiert. Die Schlüssel entsprechen den Bezeichnungen der einzelnen Graphen, welche aus dem Clocktree extrahiert wurden. So ist der Zugriff auf die Graphen gewährleistet. Der Algorithmus ist in zwei Teile aufgeteilt. Im ersten Teil werden zuerst alle Blätter des jeweiligen Graphen gefiltert und deren Strukturzahl zugeordnet. Anschließend wird im zweiten Teil den restlichen Knoten eine Strukturzahl zugeteilt. Die nummerierten Sprungmarken im Programmablaufplan verweisen auf Programmausschnitte, die den Algorithmus im Detail erläutern und im Folgenden behandelt werden.

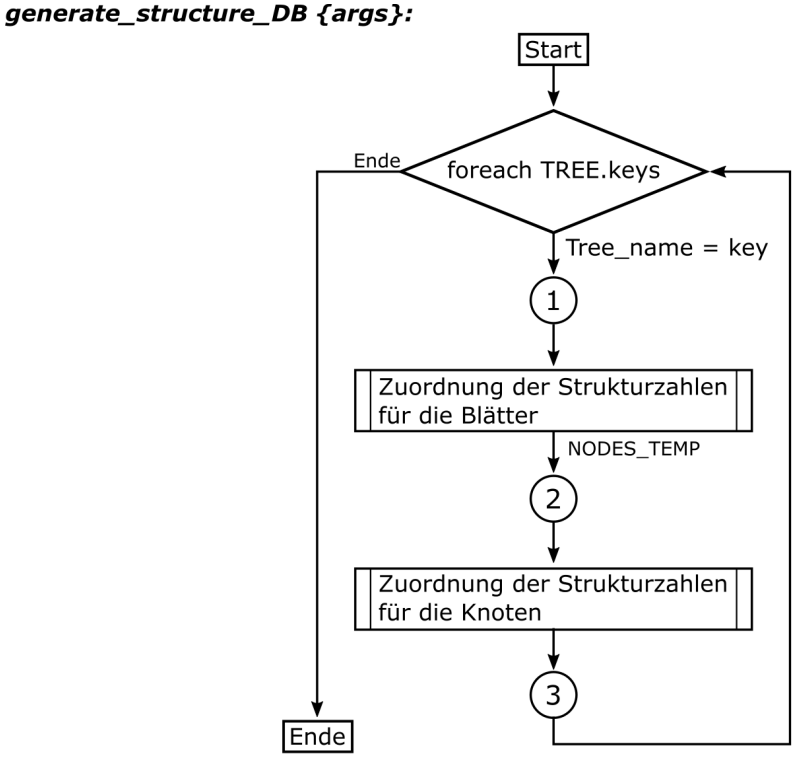


Abbildung 4.12: Funktionsweise der Prozedur `generate_structure_DB`

Der in Abbildung 4.13 dargestellte Ablaufplan zeigt die Zuordnung der Strukturzahlen für die Blätter des aktuell betrachteten Graphen. Dazu werden alle Knoten des Baumes in eine Liste gefiltert. Von dieser wird jedes Element untersucht und die Blätter werden herausgefiltert. Knoten, die keine Blätter sind, werden in einem temporären Array gespeichert, das im zweiten Teil der Analyse benötigt wird. Sobald ein Blatt identifiziert ist, wird der passende Strukturschlüssel generiert. Blätter haben generell die Strukturzahl "0". Dieser Strukturzahl wird noch der Type der Instanz, welche das Blatt repräsentiert angehängt. Eine weitere Abfrage überprüft, ob dieser Schlüssel bereits vergeben ist. Wenn der Schlüssel bereits vorhanden ist, wird dem Knoten die bekannte Strukturzahl zugewiesen. Wenn nicht, so wird eine neuer Eintrag in der Strukturdatenbasis angelegt und eine neue Strukturzahl eingepflegt. Außerdem wird bei der Strukturzahlvergabe die Anzahl der Knoten, die sich in einer Struktur befinden, in der Datenbasis hinterlegt. Über diese Information werden durch Aufruf einer Prozedur, sobald alle Struktur-IDs bekannt sind, die Bestandteile jedes Levels einer Struktur ermittelt. So ist es möglich, im Rahmen des Äquivalenz-Algorithmus die genauen Unterschiede der Strukturen zu erfassen. Eine detailliertere Beschreibung folgt im weiteren Verlauf des Abschnitts. Die Anzahl der Knoten eines Blattes beträgt immer eins.

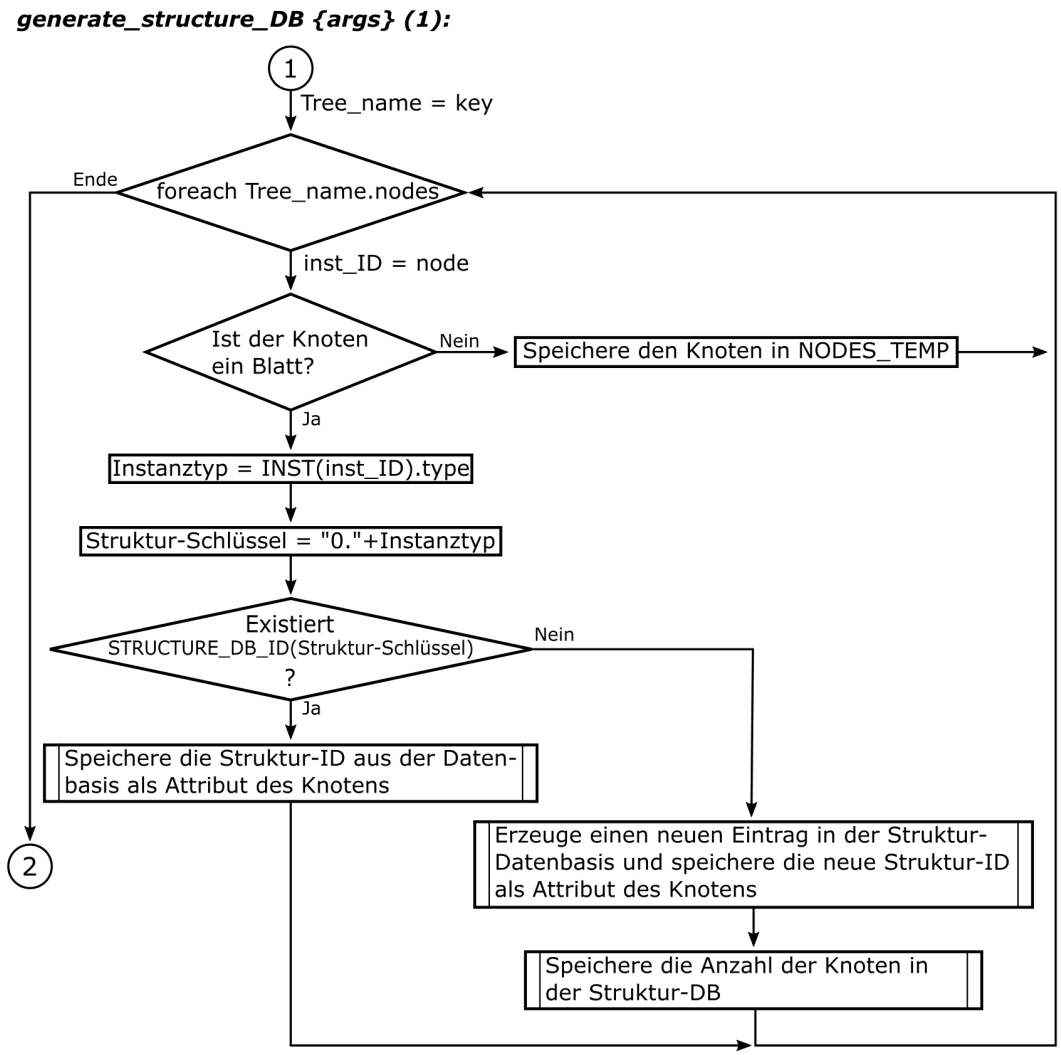


Abbildung 4.13: Zuordnung der Strukturzahlen für die Blätter

Die folgende Abbildung zeigt die Zuordnung der Struktur-IDs für die restlichen Knoten, die im Array `NODES_TEMP` gespeichert sind. Nachdem ein Knoten analysiert wurde, wird der Eintrag des Knotens im Array gelöscht. So wird das Array immer kleiner, bis es keine Einträge mehr enthält und damit alle Knoten eine Strukturzahl besitzen. Die Strukturzahlen der Children können über das Attribut "struct_id" abgefragt werden, das jedem Knoten bei der Strukturzahlvergabe angehängt wird und die Struktur-ID beinhaltet. Über die Knoten-ID kann auf die ID der Instanz referenziert, und so der Instanztyp ermittelt werden. Aus der sortierten Liste der Child-Strukturzahlen und dem Instanztyp wird der Struktur-Schlüssel erzeugt. Daraufhin wird wie im Programmablaufplan aus Abbildung 4.13 der Strukturdatenbasis-Eintrag eingepflegt. Dabei ist zu berücksichtigen, dass die Knotenzahl bei der Betrachtung dieser Knoten größer als eins ist. Die größte Knotenzahl wird in der Variablen `greatest_structure` gespeichert.

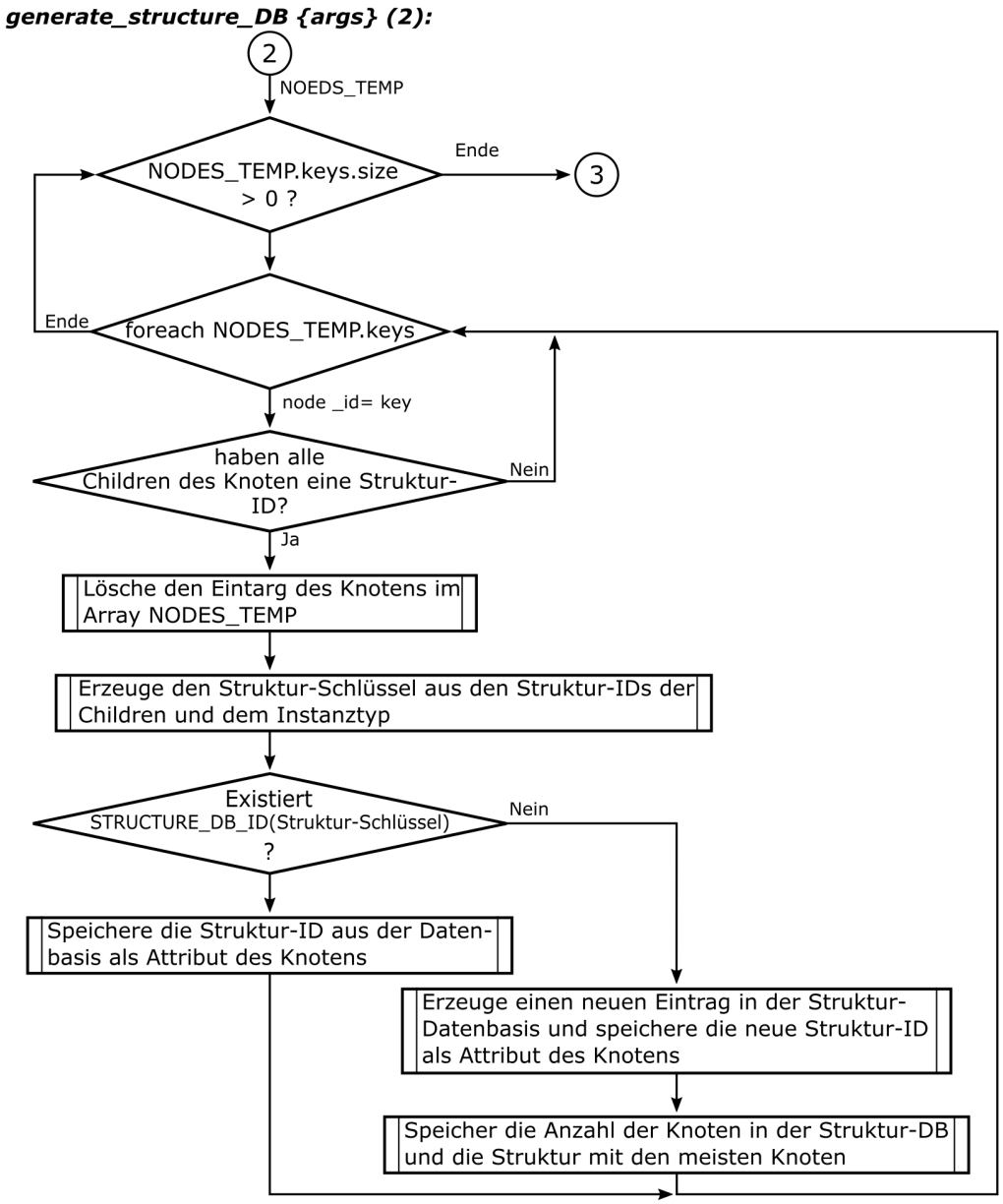


Abbildung 4.14: Zuordnung der Strukturzahlen für die Knoten

Der oben erläuterte Vorgang ist in Abbildung 4.15 anhand von zwei Beispielgraphen dargestellt. Beide Graphen teilen sich die gleiche Strukturdatenbasis, wobei die Graphen auch aus unterschiedlichen Clocktrees stammen können. Die Struktur-IDs sind in die Knoten eingezeichnet. Es ist ebenfalls zu sehen, dass die Größe der Strukturzahl von den Blättern hin zur Wurzel ansteigt. Weiterhin ist auch veranschaulicht, dass die Graphen unterschiedlich sind. Dies ist anhand der rot markierten Knoten zu erkennen.

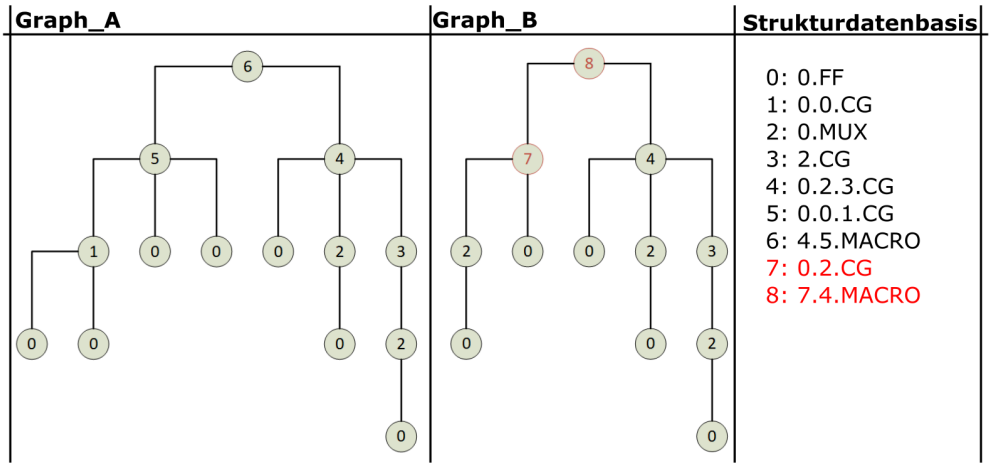


Abbildung 4.15: Zuordnung der Strukturzahlen anhand von zwei Beispielgraphen

In der oberen Abbildung wird eine Eigenschaft von Baum-Graphen sichtbar. Jede Struktur besteht aus einer bestimmten Anzahl von Knoten. Da in einem Baum keine Schleifen vorhanden sind, ist die Anzahl der Knoten einer Parent-Struktur immer größer als die Anzahl der Knoten, aus einer Child-Struktur. Beispielsweise besteht die Struktur "1" aus drei Knoten und die Struktur "5" aus sechs Knoten. Diese Eigenschaft wird in einem Algorithmus zur Bestimmung der Elemente jedes Levels verwendet. Die Nutzung dieser Eigenschaft ermöglicht eine schnelle Laufzeit dieses Algorithmus. Das Schema des Programmablaufs ist in Abbildung 4.16 dargestellt. Mithilfe einer `for`-Schleife, die bei "0" beginnt und bis zur größten Anzahl von Knoten, die in einer Struktur zu finden sind, endet, werden alle Strukturen analysiert. Das Ergebnis dieser Analysen ist eine Liste für jedes Level einer Struktur, welche alle Elemente des Levels enthält. Durch diese Listen können zwei Strukturen genau miteinander verglichen werden. Als erstes wird das erste Level einer Struktur ermittelt. Dieses entspricht dem Strukturschlüssel ohne den Instanztypen und ohne die Punkte. Die Level dieser Strukturen müssen bekannt sein, da sie eine kleinere Knotenanzahl haben als die Parent-Struktur und damit auch schon früher durch die Laufvariable der `for`-Schleife aufgerufen und analysiert wurde. Die bekannten Level können der Parent-Struktur zugeordnet werden.

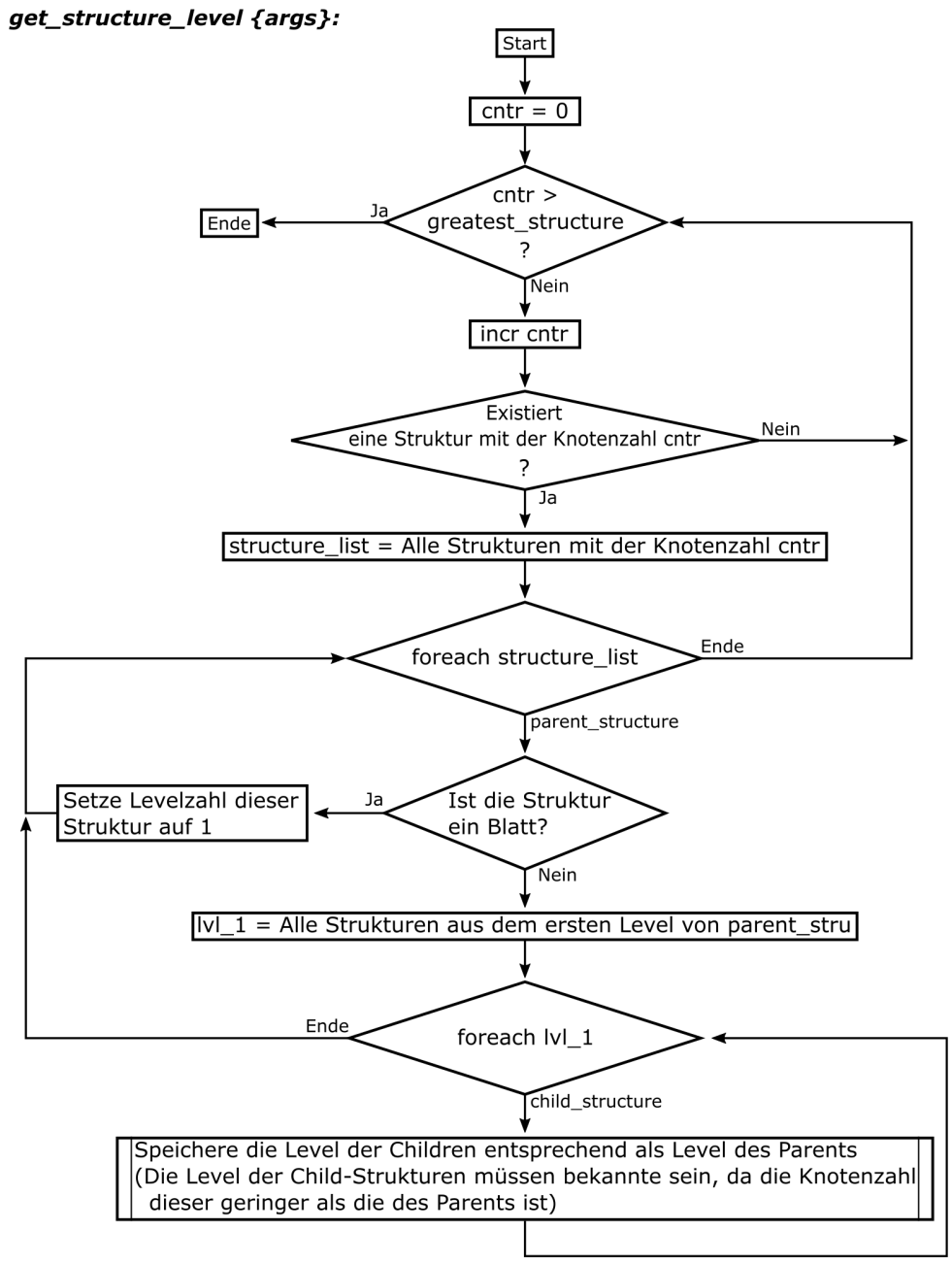


Abbildung 4.16: Analyse des Levelaufbaus der Strukturen

4.5 Der Äquivalenz-Algorithmus

Nachdem die Strukturdatenbasis erzeugt wurde, wird der Äquivalenzalgorithmus gestartet. Dies geschieht innerhalb der Sequenz des CTA-Kommandos `compare`. Dazu wird die Prozedur `compare_graphs` aufgerufen, der sechs Argumente übergeben werden. Die ersten beiden sind

die Namespace-Bezeichnungen, in denen die Informationen über den jeweiligen Clocktree hinterlegt sind. Außerdem ist die Information, ob der Vergleich detailliert oder vereinfacht stattfinden soll nicht angefügt. Diese Angabe ist in boolscher Form ausgedrückt. Zudem wird übergeben, ob der Vorgang, die Namen oder ausschließlich die Strukturen der Clocktrees berücksichtigt werden. Das letzte Argument ist der Socket-Kanal, über den die Ergebnisse an die Qt-Anwendung übertragen werden. Für diese Prozedur und alle Subprozeduren, die von dieser aufgerufen werden, existiert eine neue Datei, die wie die anderen Tcl-Dateien in die Tcl-Anwendung eingebunden werden.

Für die schnelle Analyse werden lediglich die Strukturzahlen der Wurzeln zweier Graphen miteinander verglichen und so die Äquivalenz festgestellt. Eine Prozedur, deren Funktionsweise im späteren Verlauf dieses Dokuments noch beschrieben wird, stellt zwei Struktur-IDs gegenüber, vergleicht deren Level, normiert die Werte und gibt die prozentuale Gleichheit zurück. Mithilfe dieser Prozedur wird auch der Unterschied zwischen den Graphen gemessen, wenn diese nicht isomorph sind.

Abbildung 4.17 zeigt das Schema des detaillierten Vergleichs zweier Graphen. Die Strukturzahlen sind in blau neben die Knoten geschrieben. Innerhalb der Knoten befindet sich die ID des Knotens. Der Algorithmus untersucht beide Graphen simultan. Dabei wird nach aufsteigender Level-Reihenfolge vorgegangen. In jedem Level werden dann die unterschiedlichen Strukturen ermittelt und gegenübergestellt sodass diejenigen, die sich am ähnlichsten sind, weiter verglichen werden können. Zudem wird überprüft, ob die Instanz-Namen, welche von den Knoten repräsentiert werden, gleich sind. Dieses Kriterium überwiegt die Strukturgleichheit. Zum Beispiel wird in Level 2 im Graphen A die Strukturliste: {1 3 0} ermittelt und im Graphen B die Liste: {8 9}. Wenn diese gegenüber gestellt werden, wird festgestellt, dass Struktur 1 und 8 sowie 3 und 9 miteinander verglichen werden müssen, da deren struktureller Aufbau am ähnlichsten ist. Die Namensüberprüfung wird hier nicht berücksichtigt. Das Blatt mit der ID "5" ist überzählig und kann als Ergebnis gespeichert werden. Die einzelnen Schritte sind mit den grünen und roten Pfeilen nachempfunden. Als nächstes werden die Strukturen 1 und 8 betrachtet. Dort wird festgestellt, dass die in 8 enthaltene Unterstruktur 7 nicht in Struktur 1 vorhanden ist und als Ergebnis gekennzeichnet werden kann. Anschließend werden die Knoten 4 und 4 verglichen. Diese haben im ersten Level keine direkten Unterschiede, sodass das nächste Level betrachtet werden muss. Dort ist zu sehen, dass in Graph B ein Blatt mehr vorhanden ist als in Graph A. Da nicht beurteilt werden kann, welches Blatt das Überzählige ist, werden alle als Kandidaten gekennzeichnet. Als nächstes wird überprüft, ob alle Knoten des betrachteten Levels verglichen wurden und der Algorithmus springt in das darüber liegende Level. Dies wird weitergeführt bis der Ausgangspunkt (die Wurzel) wieder erreicht ist. Die Ergebnis-Knoten sind in der Abbildung

orange gekennzeichnet.

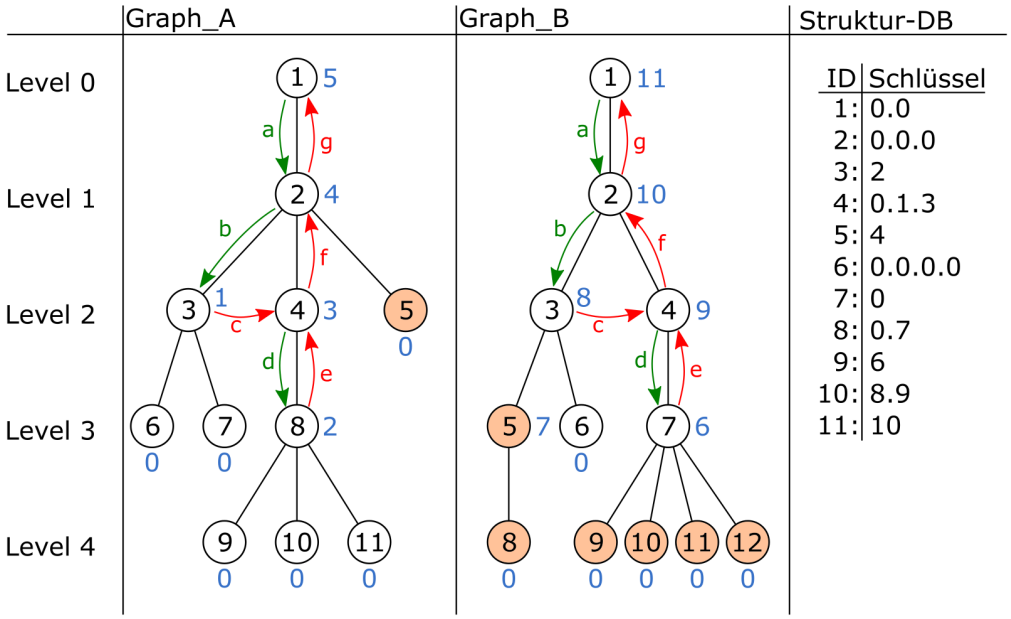


Abbildung 4.17: Ablaufschema des Äquivalenz-Algorithmus

Im Rahmen des Algorithmus ist es öfters nötig, zwei Listen miteinander zu vergleichen, um die Elemente, die gleich sind oder die, die unterschiedlich sind, zu filtern. Dazu befindet sich im Skript *CtaSubProcs.tc* eine Prozedur, die dies ermöglicht. Das Listing 4.5 zeigt die Funktionsweise der Prozedur. Wenn das optionale Argument "-inverted" übergeben wird, besteht das Ergebnis aus zwei Listen, welche die unterschiedlichen Elemente enthalten.

```
1 puts [lcompare {a b c d} {a c f}]
2 >> {a c}
3
4 puts [lcompare -inverted {a b c d} {a c f}]
5 >> {d b} {f}
```

Quelltext 4.5: lcompare

4.5.1 Gegenüberstellung von zwei Strukturlisten

Zur Feststellung, welche Strukturen miteinander verglichen werden sollen, werden zwei Strukturlisten gegenübergestellt und zu Vergleichspärchen sortiert. Dies betrifft auch die Graphen selbst. Dazu werden die Strukturzahlen der Wurzeln aller Graphen eines Clocktrees in einer Liste zusammengefasst und mit der Liste eines anderen Clocktrees verglichen. Der zentrale Bestandteil dieses Vorgangs ist der Vergleich einzelner Strukturen. Diese Funktion wird von der Prozedur `compare_structures` ausgeführt. Als Argumente werden die beiden Strukturen über-

geben. Es wird ermittelt, wie viele Knoten in einem Level zwischen den Strukturen ausgetauscht werden müssen, um gleich zu sein. Dieser Wert wird auf die Anzahl der Knoten nach diesem Austausch normiert. Der daraus resultierende Faktor, der zwischen Null und Eins liegt, wird von Eins abgezogen, sodass ein Faktor resultiert, der die Anzahl der Knoten repräsentiert, die nicht ausgetauscht werden müssen. Die folgende Formel zeigt die Gleichung. Das Ergebnis wird in Prozent angegeben.

$$Gleichheit\ des\ Levels = \left(1 - \frac{Anzahl\ der\ auszutauschenden\ Knoten}{Anzahl\ der\ Knoten\ nach\ dem\ Austausch}\right) * 100 \quad (4.1)$$

Die Berechnung wird für jedes Level summiert und durch die gesamte Level-Anzahl dividiert. So kann die prozentuale Äquivalenz der gesamten Struktur ermittelt werden. Der Vergleich der Strukturen "1" und "8" aus Abbildung 4.17 ergibt so für Level 2 100%, für Level 3 auch 100% und für Level 4 sind es 0%. Die Summe ergibt 200%. Die Anzahl der Level der größten Struktur sind drei. Wird die Summe durch diese Level-Anzahl dividiert, beträgt die Gleichheit 66,66%.

Um zwei Struktur-Listen gegenüberzustellen, wird die Prozedur `find_most_eq_structures` verwendet. Der Prozedur werden die beiden Listen als Argumente übergeben. Das Ergebnis sind zwei Listen. Die erste enthält Struktur-Pärchen, deren Partner die größte prozentuale Übereinstimmung haben. Die zweite Liste beinhaltet zwei weitere Listen, welche die Strukturen enthalten, die keine Partner-Struktur haben besitzt. Zur Gegenüberstellung der einzelnen Strukturen wird eine Matrix erstellt, in der jede Spalte und jede Zeile eine Struktur repräsentiert. Die Werte in den einzelnen Zellen entsprechen dem prozentualen Vergleich zwischen den Strukturen, welche die jeweilige Spalte und Zeile dieser Zelle repräsentieren. Für die Verwendung einer Matrix in Tcl wird das Paket "struct::matrix" verwendet. Dieses Paket bietet einige nützliche Funktionen. Weitere Informationen zur Verwendung der Bibliothek sind unter [9] und [13] zu finden. Das folgende Listing zeigt, wie die Matrix aufgestellt wird. Dabei wird das Kommando `cell` verwendet, das im Matrix-Paket enthalten ist. Mit diesem Kommando können die Zellen der Matrix gesetzt werden. In den Array `ROW` und `COL` wird die Beziehung zwischen Spaltennummer und Struktur sowie Zeilennummer und Struktur festgehalten. Dabei ist der Schlüssel die Nummer und der Wert die Strukturzahl.

```
1 set row 0; set col 0
2 foreach elemA $lstA {
3    set COL($col) $elemA
4    foreach elemB $lstB {
5       if ![info exists ROW($row)] { set ROW($row) $elemB }
6       M set cell $col $row [compare_structures $elemA $elemB "-relative"]
7       incr row
8    }
9
```

```
10    set row 0
11    incr col
12 }
```

Quelltext 4.6: Aufstellen der Matrix

Die Anzahl der Struktur-Pärchen entspricht der Länge der kürzeren Strukturliste, die an die Prozedur übergeben wurde. Der folgende Quellcode zeigt, wie diese Pärchen ermittelt werden können. Dabei wird die `for`-Schleife so oft wiederholt, bis die maximale Pärchenzahl erreicht ist. Die dritte Zeile liefert den höchsten Wert aus der Matrix. In den Zeilen fünf bis neun werden die Zeilen- und Spaltennummer ermittelt, die zu dem zuvor erfassten Wert gehören. Daraufhin können über die Zeile/Struktur und Spalte/Struktur Beziehungen der Arrays `COL` und `ROW` die Struktur-Pärchen ermittelt werden. Am Ende werden die Elemente in dieser Zeile und dieser Spalte auf "-1" gesetzt, sodass diese von den weiteren Analysen ausgeschlossen sind.

```
1  for {set i 0} {$i < $comp_variations} {incr i} {
2     #erfasse die Zelle mit dem höchsten Wert
3     set max_val [tcl::mathfunc::max {*}[join [M get rect 0 0 end end]]]
4     #ermittel die Zeilen und Spaltennummer zu diesem Wert
5     set cell [join [M search all $max_val]]
6     #erfasse die zugehörigen Strukturen
7     set col [lindex $cell 0]
8     set row [lindex $cell 1]
9     lappend return_lst [list $COL($col) $ROW($row)]
10    #setze alle Werte dieser Zeilen und Spalten auf -1
11    M set column $col $mone_column
12    M set row $row $mone_column
13 }
```

Quelltext 4.7: Aufstellen der Matrix

Abbildung 4.18 zeigt das Schema des Strukturlisten-Vergleichs. Die Matrix basiert auf den Strukturen, die im zweiten Level in Abbildung 4.17 dargestellt sind. Die Werte in den Zellen sind mithilfe der Formel 4.1 berechnet worden. Zu sehen ist, wie die größten Werte ermittelt und anschließend die bereits betrachteten Zeilen für die weitere Analyse ausgeschlossen werden, bis alle Ergebnisse vorliegen. Das Ergebnis zeigt die zu vergleichenden Strukturen. Die Listen für alleinstehende Strukturen, die keinen Vergleichspartner haben sind leer.

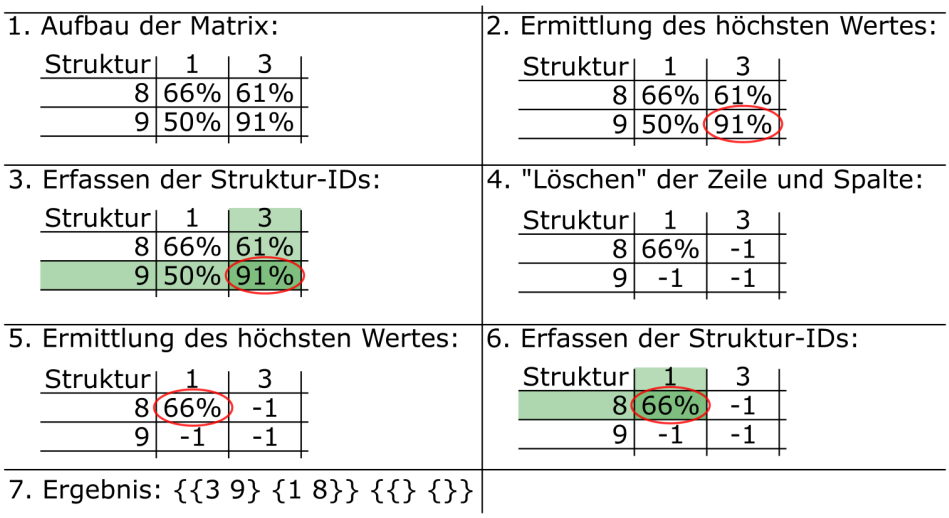


Abbildung 4.18: Schema der Gegenüberstellung von Strukturlisten

4.5.2 Vergleich von Graphen

Als erster Schritt im Äquivalenzalgorithmus wird überprüft, welche der Graphen miteinander verglichen werden sollen. Dazu werden die Struktur-IDs der Wurzeln ermittelt und -wie in Abschnitt 4.5.1 erläutert-, miteinander verglichen. Wenn alle Graphen isomorph sind, sind beide Clocktrees identisch und der Prozess kann beendet werden. Weisen die Graphen Ungleichheiten auf, so wird geprüft, ob der Vergleich schnell oder detailliert stattfinden soll. Bei dem schnellen Äquivalenzcheck wird ähnlich wie im vorigen Abschnitt der prozentuale Gleichheitswert ermittelt und in der Konsole ausgegeben. Wenn eine gründliche Analyse der Clocktrees stattfinden soll, wird ein Algorithmus angewendet der im Folgenden erläutert wird.

Der Programmablauf folgt dem Schema aus Abbildung 4.17 zu Beginn dieses Kapitels. Als erstes werden die Wurzeln der Graphen, die keinen Vergleichspartner haben, als Ergebnis in einem Array abgelegt. In diesen Arrays werden alle Elemente gespeichert, die überzählig sind oder sich in einem unvergleichbaren Pfad befinden. Beide Ereignisse sind im Folgenden als Fehlertyp beschrieben. Zu den Einträgen wird ein Vermerk hinterlegt der auszeichnet, zu welchem Clocktree die Knoten gehören und welcher Fehlertyp ihnen zugeordnet ist. Der Algorithmus basiert darauf, dass zwei Graphen gegenübergestellt werden und nur Pfade betrachtet werden, deren Knoten unterschiedliche Strukturen haben. Dabei wird die Variable `vert_lvl_ctr` als Zähler verwendet, um schrittweise die Level analysieren zu können. Mithilfe des zweiten Zählers `hor_lvl_ctr` können die einzelnen Elemente in einem Level angesprochen werden. Der Algorithmus filtert für jedes Level diejenigen Knoten heraus, die strukturelle Unterschiede aufweisen. Für jedes Level und jeden Graphen wird dazu eine Liste angelegt. Die unterschiedlichen Knoten werden dann so sortiert, dass die Vergleichs-Paare den gleichen Index in den Listen haben. Die folgende Ab-

bildung zeigt anhand von zwei Beispiellisten den Ablauf, um diesen zu veranschaulichen.

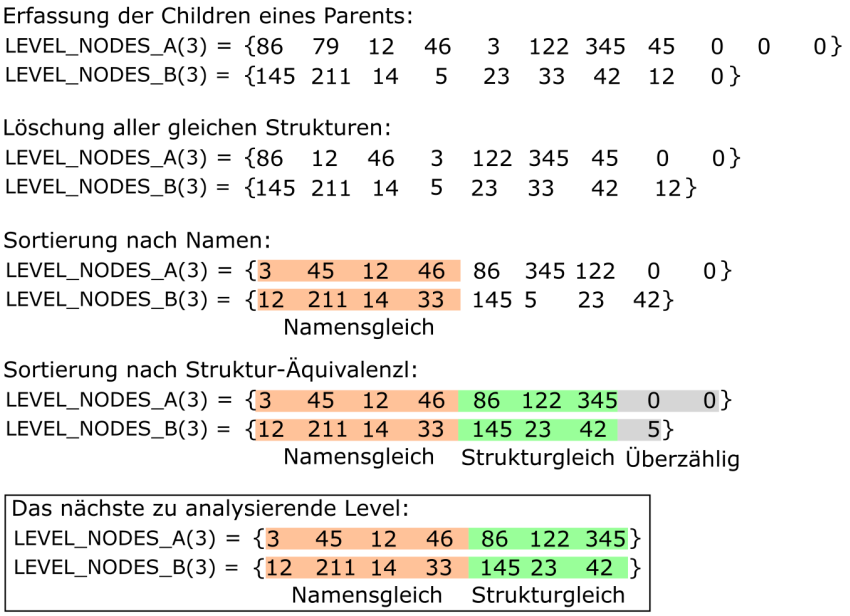


Abbildung 4.19: Anpassung der Knoten-listen von zwei Graphen im gleichen Level

Der Programmablaufplan aus Abbildung 4.20 zeigt, wie die überzähligen Knoten ermittelt werden. Die farbigen Felder verweisen auf die obigen Schritte. Nach der Sortierung und der Entfernung von Ergebnis-Knoten wird ein neues, zu analysierendes Level angelegt. Daraufhin wird durch die Inkrementierung des Zählers `vert_lvl_ctr` das nächste Level analysiert. Dort wird die Analyse mit dem ersten Knoten begonnen. Wenn dieser Knoten kein Blatt ist, wird der vertikale Zähler erneut inkrementiert. Sind keine weiteren Knoten vorhanden, wird der Zähler dekrementiert, sodass der Algorithmus ein Level aufsteigt, an welcher Stelle der horizontale Zähler inkrementiert wird. So kann der zweite Knoten in diesem Level analysiert werden. Sobald dieser Zähler so groß ist wie die Anzahl der Knoten im Level, wird das darüber liegende Level weiter analysiert. Dieser Vorgang wiederholt sich, bis der vertikale Zähler den Wert Null annimmt.

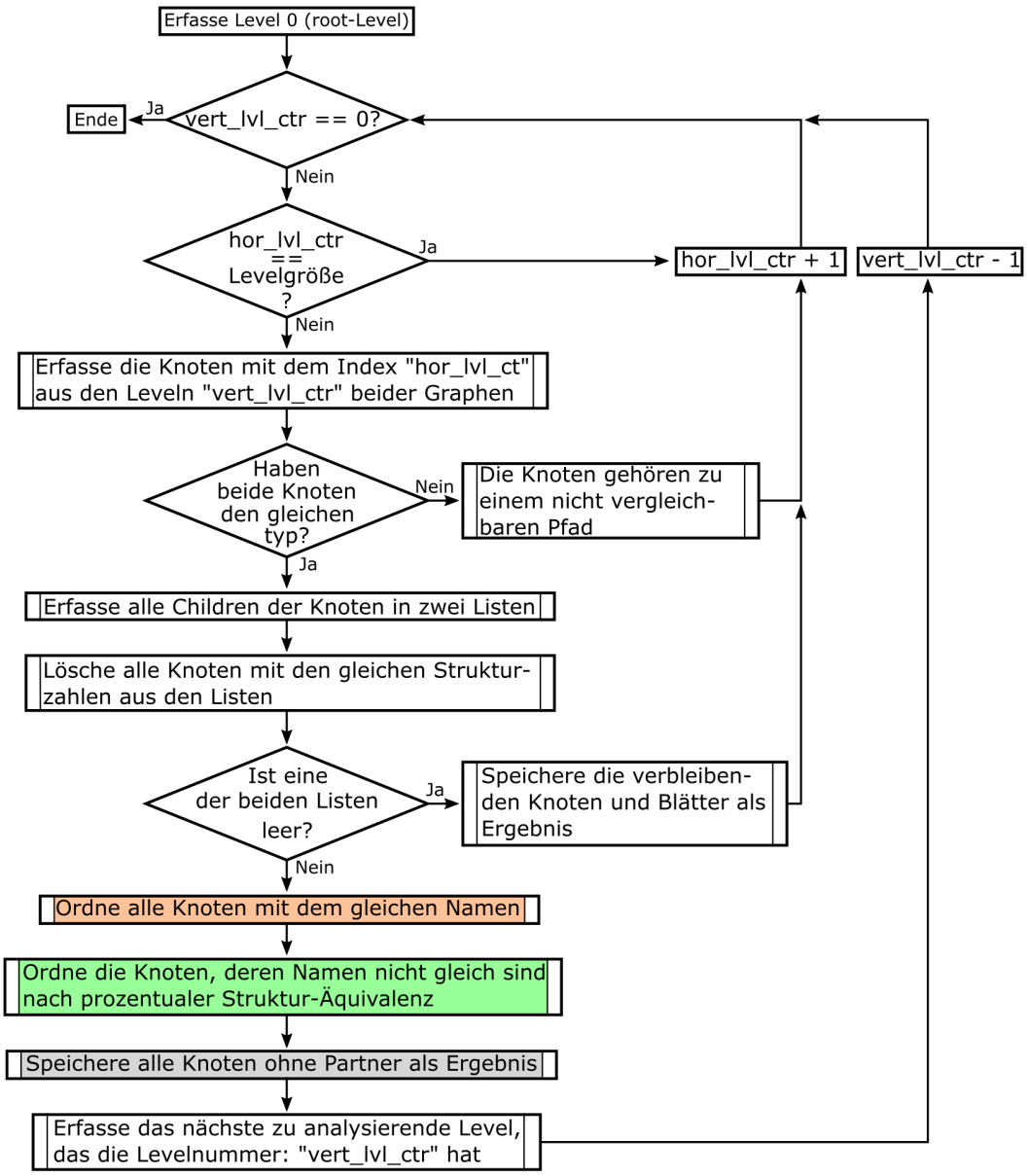


Abbildung 4.20: Programmablaufplan des Graphenvergleichs

Abschließend findet die Übertragung der Ergebnisse statt. Diese werden per TCP/IP-Verbindung an die Qt Anwendung transferiert. Von den Elementen, die sich in einem unvergleichbaren Pfad befinden, wird nur das erste erkannt. Mithilfe des tcl-tree-Kommandos `children` können alle Instanzen in einem solchen Pfad erfasst werden. Durch die Referenz in der Datenbasis zu den an den Instanzen angeschlossenen Netzen können auch diese ermittelt werden. Bei der Übertragung dieses Elementes wird auch die entsprechende Farbe zur Kolorierung und dem ersten Element das Schlüsselwort "tool_log" übergeben. Durch das Schlüsselwort wird festgelegt, das nur diese Instanz in der Konsole des Comparetools ausgegeben wird. Das gleiche gilt auch für Strukturen,

die aus mehreren Children bestehen. Die zu transferierenden Datensätze beinhalten außerdem die Bezeichnung des Clocktrees und die Information, ob es sich um ein Netz oder eine Instanz handelt.

Wenn im Comparetool der Kontrollkasten für den Namensvergleich aktiviert ist, wird vor der Übertragung geprüft, ob die Bezeichnung für diese Instanz auch im zweiten Clocktree vorhanden ist. In dem Fall, dass dies zutrifft, findet keine Übertragung statt. So kann die Anzahl von überzähligen Kandidaten eingegrenzt werden. Voraussetzung dafür ist jedoch, dass die Bezeichnungen der Instanzen sich nicht geändert haben.

5 Ergebnisdarstellung

In diesem Kapitel werden die Ergebnisse vorgestellt, die der Äquivalenz-Algorithmus liefert und wie diese in der GUI dargestellt werden. Abbildung 5.1 zeigt die fertige Benutzeroberfläche. Der linke obere Teil ist das Comparetool, dessen Größe anpassbar ist. Darunter ist die Minimap zu sehen, die bereits Teil der Demo-Applikation war. Auf der rechten Seite befinden sich das Schematic-Widget und die Tcl/API-Konsole.

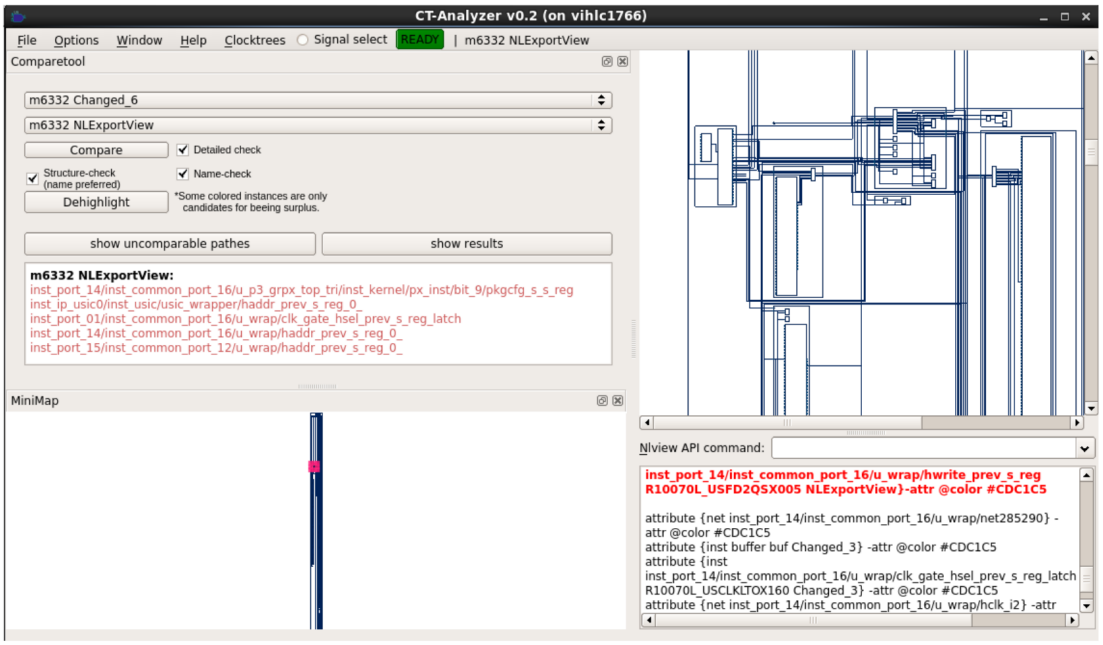


Abbildung 5.1: GUI der fertigen Programms

Die Kolorierung der nicht vergleichbaren Pfade ist in Abbildung 5.2 dargestellt. Um den Buffer, der den Unterschied hervorruft, besser sehen zu können, ist dieser Teil der Schaltung vergrößert dargestellt. Auf der rechten Seite in Clocktree_A ist der Buffer nicht vorhanden. Der Äquivalenz-Algorithmus stellt fest, dass hier zwei unterschiedliche Instanz-Typen miteinander verglichen werden. Der Buffer auf der linken und das Clockgate auf der rechten Seite. Die nachfolgende Struktur kann in diesem Fall nicht weiter verglichen werden und wird grau eingefärbt.

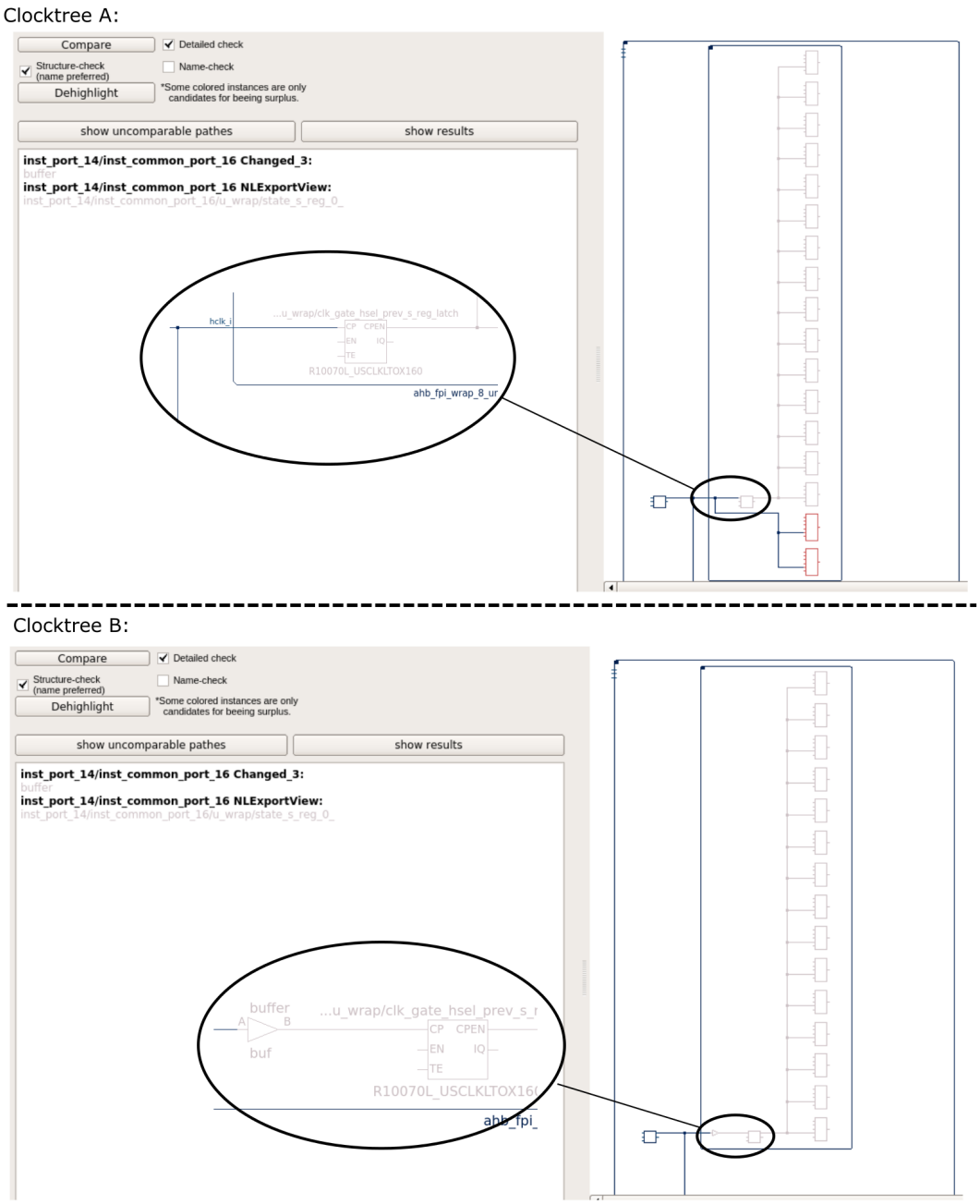


Abbildung 5.2: Kolorierung eines nicht vergleichbaren Pfades

Abbildung 5.3 zeigt die beiden unterschiedlichen Clocktrees "Clocktree_A" und "Clocktree_B". Es ist zu sehen, dass auf der rechten Seite ein Clockgate und mehrere Flipflops vorhanden sind, die auf der linken Seite fehlen. Die Instanzen und das Netz, welches die Elemente verbindet, sind farbig markiert.

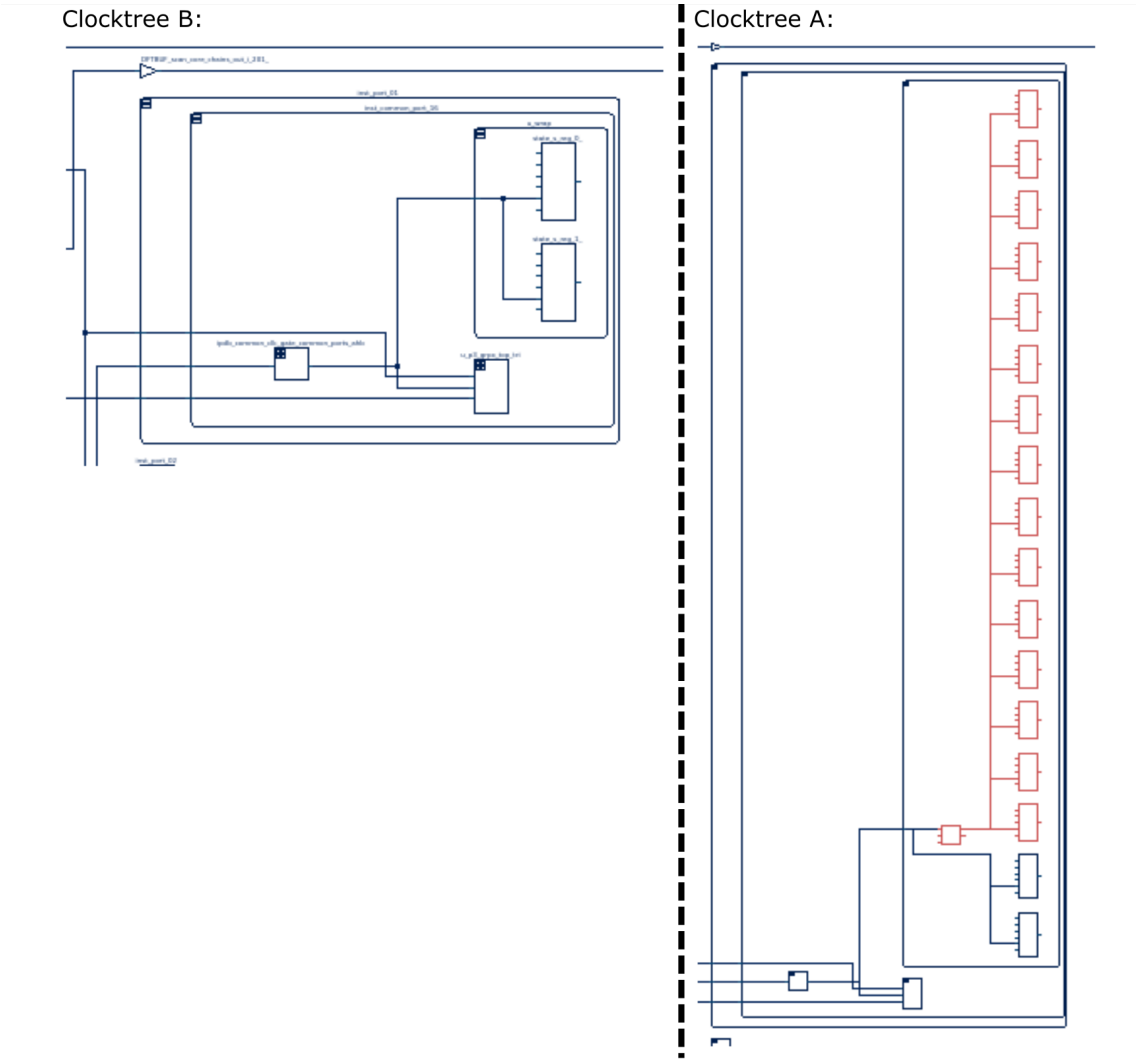


Abbildung 5.3: Kolorierung einer Struktur mit mehreren Children

In Abbildung 5.4 sind zwei unterschiedliche Clocktrees zu sehen, die mit unterschiedlichen Detail-Optionen verglichen wurden. Im oberen Teil der Abbildung auf der linken Seite ist der Namensvergleich ausgeschaltet. So werden alle Instanzen markiert, die als überzählig in Frage kommen. Durch den Namensvergleich kann festgestellt werden, welche der fünf Instanzen nicht in Clocktree_A vorhanden ist. Es ist jedoch möglich, dass die Bezeichnungen der Elemente sich ändern, wenn beispielsweise ein anderer Flipflop-Typ ausgewählt wurde. Die betroffenen Elemente sind durch rote Markierungen gekennzeichnet.

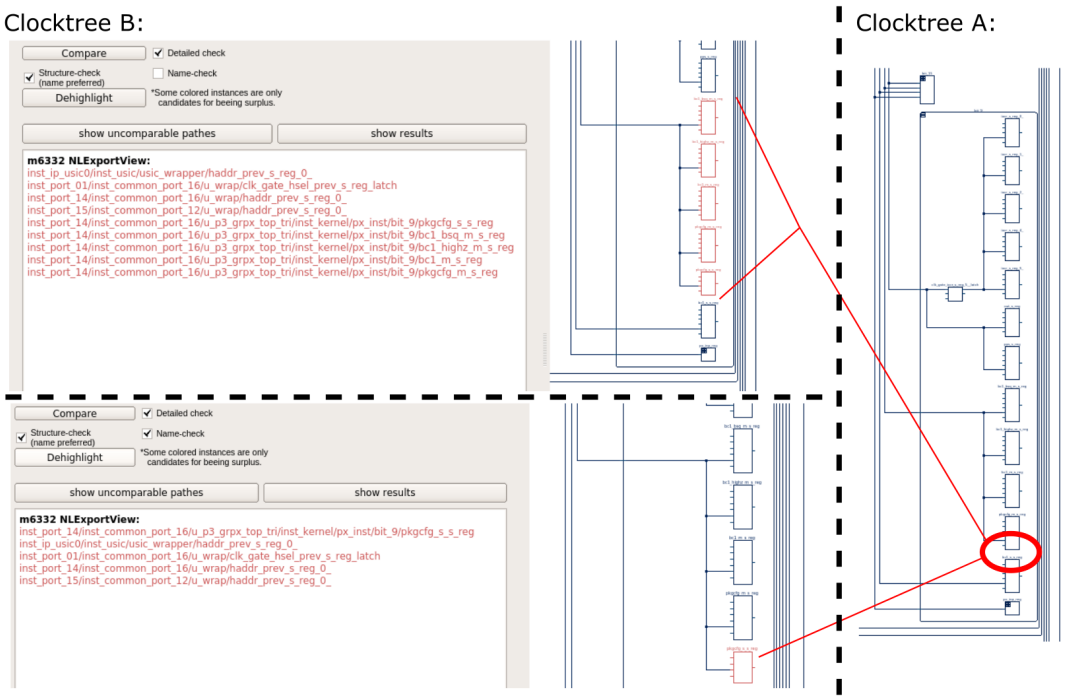


Abbildung 5.4: Der Unterschied zwischen den Detail-Optionen im Comparetool

6 Zusammenfassung und Ausblick

In einem Projekt zur Analyse von Clocktrees komplexer SoCs bei der Firma *EPOS* wurde ein Tool entwickelt, mit dem es möglich ist, Clocktrees zu visualisieren, übersichtlicher darzustellen und zu analysieren. Aufgabenstellung dieser Masterarbeit war es den *Clocktree-Analyzer* zu erweitern. Dazu war eine Einarbeitung in das bestehende Programm notwendig. Die zur Visualisierung notwendigen Daten können aus den bei *EPOS* verwendeten Tools extrahiert werden. Das entstandene Programm trägt die Bezeichnung *Clocktree-Analyzer* und basiert auf zwei unterschiedlichen Applikationen. Eine davon beruht auf der Skriptsprache Tcl. Der Vorteil dieser Sprache liegt in ihrer Einfachheit und der weiten Verbreitung in der EDA Industrie. Die zweite Applikation basiert auf der C++ Bibliothek Qt und beinhaltet die Schematic-Viewer-Engine *NLView* der Firma *Concept Engineering*. Diese Engine wird zur Anzeige der Schematics verwendet.
Die Qt-Applikation und die Tcl-Anwendungen kommunizieren über eine TCP/IP-Verbindung. Diese Qt-Applikation wird für die Übertragung spezieller CTA-Kommandos und großer Datenmengen verwendet. Die CTA-Kommandos werden von der Qt-Applikation an die Tcl-Anwendung gesendet und dort interpretiert. Daraufhin werden in Tcl realisierte Algorithmen durchlaufen und die Ergebnisse zurück an die Qt-Applikation übertragen, wo sie in der Schematic-Viewer-Engine *NLView* angezeigt werden.
Die Tcl-Anwendung verfügt über eine interne Datenbasis, welche durch das Einlesen von sogenannten NLV-Dateien erzeugt wird. Das Format dieser Dateien besteht aus API-Kommandos, die von der *NLView*-Engine verwendet werden. Das Einlesen dieser Dateien in die Tcl-Datenbasis wird über ein CTA-Kommando gesteuert. Der Vorgang wird durch einen Menüeintrag in der GUI eingeleitet.

Die Erweiterung, welche im Rahmen dieser Masterarbeit implementiert wurde, ergänzt den *Clocktree-Analyzer* um eine Komponente zum Vergleich von Clocktrees. Dazu war neben dem Verständnis des Softwareaufbaus des bestehenden Programmes auch der Umgang mit *NLView* relevant. Die Komponente wird als *Comparetool* bezeichnet. Mithilfe der Benutzerschnittstelle, welche in die Qt-Applikation integriert wurde, ist es möglich zwei Clocktrees auszuwählen und in verschiedenen auswählbaren Modi miteinander zu vergleichen. Dabei ist es möglich eine schnelle Analyse durchzuführen, welche eine prozentuale Beurteilung über die Äquivalenz liefert. Ein weiterer Modus bietet einen detaillierten Vergleich der Clocktrees. Dabei werden al-

le Elemente, die einen Unterschied verursachen, im Schematic eingefärbt und in einer Konsole ausgegeben. Für die detaillierte Analyse kann optional ausgewählt werden, ob die Ergebnisse einem Namenscheck unterzogen werden, bei dem überprüft wird, ob die Bezeichnungen der gefundenen Elemente im jeweils anderen Clocktree vergeben sind. So ist eine Spezifizierung der Ergebnisse möglich. Da sich die Bezeichnungen ändern können, liefert diese Option aber keine exakten Befunde, sodass der strukturelle Vergleich unerlässlich für ein korrektes Resultat ist. Das neu eingeführt CTA-Kommando "compare" sorgt dafür, dass in der Tcl-Anwendung die Algorithmen ausgeführt werden.
Der Strukturcheck basiert auf der Graphentheorie. Dazu werden die ausgewählten Clocktrees in der Tcl-Anwendung zu Baum-Graphen umgewandelt. Diese Graphen können in ihre einzelnen Strukturen aufgeteilt werden. Jede dieser Strukturen besteht aus einer Anordnung von Folgeknoten und bekommt eine ID zugewiesen. Die Strukturen werden mitsamt ihrer IDs in einer Clocktree unabhängigen Struktur-Datenbasis hinterlegt. Diese Datenbasis beinhaltet alle Strukturen, die in jedem Clocktree gefunden wurden, der einem Vergleich unterzogen wurde. Für jede neue Struktur wird ein neuer Eintrag in der Datenbasis angelegt. Bereits bekannte Strukturen erhalten die vorhandenen IDs. Auf diese Weise können Strukturen, die in einem Clocktree vorhanden sind und in einem anderen nicht, ermittelt werden. Damit ist es möglich die strukturellen Unterschiede in den Clocktrees festzustellen und basierend auf den so gewonnenen Informationen die *Constraints* zu konkretisieren.

Mithilfe des Äquivalenzalgorithmus ist es möglich Strukturunterschiede in den Clocktrees zu finden. Einige Strukturen können jedoch nicht im Detail analysiert werden. So werden nur die gesamte Struktur und alle Instanzen, die durch diese repräsentiert werden, als Ergebnis erfasst. Solche Strukturen werden als unvergleichbare Pfade bezeichnet und im Schematic grau gekennzeichnet. Ein weiterer Algorithmus, welcher in der Lage ist diese Pfade zu analysieren und die einzelnen Instanzen zu erkennen, wäre eine sinnvolle Ergänzung. Besonders die Implementierung von Buffern während der *Clocktree-Synthese* verursacht die Entstehung solcher unvergleichbaren Pfade. Die Ignorierung dieser Buffer während der Strukturerfassung würde es ermöglichen, Clocktrees auch nach der CTS sinnvoll vergleichen zu können. Darüber hinaus würde die Kompression von Bufferketten, die während der CTS implementiert werden, die Übersichtlichkeit des Clocktrees verbessern. Dabei könnten mehrere Buffer in eine Pseudo-Hierarchie zusammengefasst werden. Eine solche Erweiterung stellt eine sinnvolle Ergänzung dar, um Clocktrees auch nach der CTS benutzerfreundlich betrachten zu können.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] Peter Ashenden. *Digital Design - An Embedded System Approach Using VHDL*. 1. Auflage. Elsevier, 2008. ISBN: 978-0-12-369528-4.

[2] Lene Baur. *Königsberger Brückenproblem*. 2009. URL: http://www.mathematik.uni-marburg.de/~bschwarz/Sem_09W_files/02%20Lene%20Baur%20-%20K%C3%B6nigsberger%20Br%C3%BCckenproblem%20-%20Ausarbeitung.pdf.

[3] Ken Jones Brent B. Welch. *Practical Programming in Tcl and Tk*. 1. Auflage, 2. Druck. Prentice Hall, 2003. ISBN: 0-13-038560-3.

[4] Concept Engineering. *NLView-Qt Documentation*. 2017.

[5] Mark Summerfield Jasmin Blanchette. *C++ GUI Programming with Qt 4*. 1. Auflage, 3. Druck. Prentis Hall, 2009. ISBN: 978-0-13-235416-5.

[6] Desmond Kabus. *Das Königsberger Brückenproblem*. 2011. URL: http://www.friderizianer.de/enzyklopaedie/kaliningrad-bruecken/.

[7] Hubert Kaeselin. *Top-Down Digital VLSI Design - From Architctures to Gate-Level Circuits and FPGAs*. 1. Auflage. Elsevier, 2015. ISBN: 978-0-12-800730-3.

[8] Prof. Dr. Wolfgang Konen. *Graphentheorie*. 2018. URL: http://www.gm.fh-koeln.de/~konen/Mathe2-SS/ZD2-Kap09.pdf.

[9] Andreas Kupries. *struct::matrix(3tcl) 2.0.1 struct Tcl Data Structures*. 2002. URL: https://tools.ietf.org/doc/tcllib/html/matrix.html.

[10] Andreas Kupries. *struct::tree(n) 2.1.1 tcllib Tcl Data Structures*. 2012. URL: https://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/struct/struct_tree.html#28.

[11] Jens Lienig. *Layoutsynthese elektronischer Schaltungen*. 1. Auflage. Springer-Verlag, 2006. ISBN: 978-3-540-29627-0.

[12] Rolf Niedermeier. *Ausgewählte Graphalgorithmen*. 2004. URL: http://theinf1.informatik.uni-jena.de/publications/agscript.pdf.

[13] Oliver Scholl. *Einführung in Tcl/Tk*. 4. Auflage. Oliver Scholl, 2017. URL: http://www.tcltk.info/einfuehrung_in_tcl_tk_buch.pdf.