



**Systemhaus für intelligente
EDV-Anwendungen**

Richard-Wagner-Straße 91
67655 Kaiserslautern
Telefon +49 631 36 30 15-0
Telefax +49 631 36 30 15-33
E-Mail: sieda@sieda.com
WWW: <http://www.sieda.com>

ConSolve **Anwenderhandbuch**

Peter Lenhard
Dr. Harald Meyer auf'm Hofe, Enno Tolzmann

SIEDA H-2/0.95-1

Technischer Bericht



Inhaltsverzeichnis

1	Einleitung:	1
2	Das Prinzip ConSolve: Eine erste Einführung	3
2.1	Beispiel-Problem	3
2.2	Vorteile von ConSolve	4
2.3	Problembeschreibung in ConStruct	4
2.4	Codegenerierung	6
2.5	Integration in eine Anwendung	6
2.6	Besondere Merkmale	8
2.7	Zusammenfassung	9
3	Zum Einstieg: Ein genauer Blick auf ein Beispiel	10
3.1	„Lateinisches Quadrat“: mathematische/logische Formulierung	10
3.2	„Lateinisches Quadrat“: Formulierung in der Beschreibungssprache	11
3.3	Von der Beschreibung zur Lösung: Codegenerierung mit <i>cssh</i>	14
4	Zum Kennenlernen: Integration und Kommunikation	16
4.1	Codegenerierung mit <i>cssh</i> oder <i>cswin</i>	16
4.1.1	Vorbedingungen	16
4.1.2	Aufruf	17
4.1.3	Ergebnis	17
4.1.4	Verwendung der Dateien	17
4.2	Die Verwendung exportierter Konzepte unter C++	18
4.2.1	Der Iterator	18
4.2.2	Der Export von Prädikaten, Konzepten und Methoden	18
4.2.3	Namenskonventionen	19

4.3	Zusätzliche Bedingungen einfügen	20
4.4	Manuell entstandene Lösungsvorschläge	21
4.5	Beobachter im ConSolve-System	21
4.5.1	Die Funktionen der Beobachter	22
4.6	Suchbaum schreiben	23
4.7	Fehlerbehandlung	24
4.7.1	Die Typen der Ausnahmen	24
4.7.2	Fehlernummern und Kurzbeschreibung	25
4.7.3	Fehlerbehandlung in der C-Schnittstelle	26
4.7.4	<i>Callback</i> -Mechanismus als Alternative	26

1 Einleitung:

Dies ist das „Anwenderhandbuch zu ConSolve“. Es setzt Kenntnisse der Logik und der Programmiersprache C++ voraus. Über dieses Handbuch hinaus gibt es ein detaillierteres „Referenzhandbuch ConSolve/ConStruct“ und eine ausführliche Dokumentation zur „Schnittstelle von ConStruct zu den Programmiersprachen C und C++“.

Die Ziele dieses Handbuches sind:

1. Einem Interessenten die deskriptive Herangehensweise zu erläutern und den Leistungsumfang des Systems zu vermitteln.
Dazu empfiehlt sich die Lektüre des Handbuches zumindest bis Kapitel 3.
2. Einem mit der Constraint-Technik vertrauten Nutzer, der mit dem ConSolve-System noch nicht vertraut ist, den Leistungsumfang des ConSolve-Systems darzustellen und die Benutzung zu ermöglichen.
Das Kapitel 2 kann dabei übergangen werden; Kapitel 3 bis zum Abschnitt 3.2 ebenfalls.
3. Einem Anwender des ConSolve-Systems als Nachschlagewerk zu dienen. Neben Glossar und Index sind hierzu die aufgelisteten Befehle des Werkzeuges *cssh* (bzw. *cswin*) und die Beschreibung des ConStruct-Sprachumfangs (Kapitel 7 und 8) hilfreich.
Fortgeschrittene Anwender sollten zusätzlich die Referenzdokumentation heranziehen.

Das Kapitel 2 gibt einen kurzen Abriss des ConSolve-Systems. Es zeigt an einem ersten Beispiel die Arbeitsweise von ConSolve und die Stärken des Systems. Dieses Kapitel kann für sich genommen als Kurzbeschreibung dienen.

Das Kapitel 3 erläutert ausführlich das erste Beispiel und führt dabei einen Teil des ConStruct-Sprachumfangs ein. Die Codegenerierung mit dem Werkzeug *cssh* zur Integration in C++ wird kurz angesprochen.

Das Kapitel 4 erläutert die Integration des ConSolve-Systems in Anwendungsprogramme anhand der Sprache C++. Es beschreibt die Codegenerierung und die Verwendung exportierter Konzepte in einer Programmierumgebung. Darüber hinaus erläutert das Kapitel die Abfrage von Lösungen mittels Iteratoren und zeigt weitere Möglichkeiten der Interaktion zwischen Anwendung und ConSolve-System. Beispiele dafür ist das Einfügen zusätzlicher Bedingungen zur Laufzeit oder die Erzeugung einer Suchbaumdatei zur späteren Analyse.

Das Kapitel 5 beschreibt die Modellierung ausgewählter Problemstellungen in ConStruct und führt weitere ConStruct-Sprachelemente ein.

Das Kapitel 6 legt dar, auf welche Weise das ConSolve-Laufzeitsystem den Raum der potentiellen Lösungen einschränkt und wie darin nach Lösungskandidaten gesucht wird. Die Parameter zur Steuerung der Suche werden beschrieben.

Das Kapitel 7 beschreibt die wichtigsten Leistungsmerkmale, die die Werkzeug *cssh* und *cswin* neben der einfachen Codegenerierung noch bieten. Stichworte dazu sind: weitere Optionen zum parametrisierten Aufruf, Umgang mit ConStruct-Modulen, interaktive Problemlösung ohne Anwendungsprogramm.

Das Kapitel 8 beschreibt alle Schlüsselwörter und Sprachelemente von ConStruct.

2 Das Prinzip ConSolve: Eine erste Einführung

Das ConSolve-System löst kombinatorische Optimierungsprobleme. Dazu werden die Problemstellungen in der speziellen Beschreibungssprache „ConStruct“ formuliert und dann dem ConSolve-Laufzeitsystem übergeben. Das System sucht Lösungen mittels Techniken aus der „Künstlichen Intelligenz“ (KI).

Die Sprache ConStruct wurde entworfen, um logische und kombinatorische Probleme möglichst einfach formulieren zu können. Sie wurde also an eine spezielle Aufgabe angepasst, weshalb man sie allgemein als „domänenspezifische Sprache“ bezeichnen kann. Auf diese Weise kann die Formulierung sehr nahe an der Problemstellung gehalten werden, sie ist übersichtlich und die Arbeit daran wenig fehlerträchtig. Darüber hinaus ist es möglich, relativ schnell zu einer Formulierung für ein Problem zu gelangen.

In der Sprache ConStruct werden keine Lösungswege beschrieben, sondern lediglich Problemstellungen formuliert. Die Auswahl der zur Problemlösung verwendeten Algorithmen geschieht durch das ConSolve-Laufzeitsystem. Somit handelt es sich zugleich um eine „deklarative Sprache“.

Die Arbeit mit dem ConSolve-System unterscheidet sich wesentlich von der Arbeit mit üblichen (algorithmischen) Programmiersprachen und ist für viele Entwickler sicherlich ungewohnt. Die deklarative Herangehensweise an logische Probleme hat jedoch große Vorteile, die schnell deutlich werden. Das ConSolve-System lässt sich außerdem einfach und schnell in übliche Programmierumgebungen integrieren.

2.1 Beispiel-Problem

Die Mächtigkeit der domänenspezifischen deklarativen Sprache ConStruct und die Integration in eine C++-Anwendung demonstrieren wir an einem einfachen Beispiel, dem sogenannten „Lateinischen Quadrat“. Hierbei handelt es sich um eine mathematische Knobelei, die vielleicht einigen Leserinnen und Lesern schon bekannt ist; ein Quadrat mit n mal n Feldern soll so mit Zahlen zwischen 1 und n gefüllt werden, dass folgende Bedingungen zutreffen:

- In jeder Zeile kommen die Zahlen 1 bis n jeweils genau einmal vor.
- In jeder Spalte kommen die Zahlen 1 bis n jeweils genau einmal vor.
- Auf beiden Diagonalen kommen die Zahlen von 1 bis n jeweils genau einmal vor.


```

// "Summe" ist die Summe der Zahlen von 1 bis Groesse
Summe := forall i:{1..Groesse :Z} apply + to (i);
concept Quadrat
begin
  // Ein "Quadrat" besteht aus (Groesse) Zeilen
  // mit je (Groesse) Spalten und darin
  // Einträgen zwischen 1 und (Groesse).
  entail // stelle sicher (Prädikat folgt)
    L[{1..::Groesse :Z}, {1..Groesse :Z} : {1..::Groesse :Z}];
  // "eintrag" wird für den Zugriff
  // von außen passend definiert.
  export
    eintrag(zeile:{0.. :Z}, spalte:{0.. :Z}) :T
      := this[zeile,spalte];
end Quadrat;
// Neues Konzept: ein Lateinisches Quadrat
concept LateinischesQuadrat
begin
  // ist ein Quadrat
  inherit Quadrat;
  // Für jede Zeile gilt:
  // Jeder Eintrag ist unterschiedlich.
  entail
    forall zeile:{1..::Groesse :Z}
      entail forall spalte:{1..Groesse :Z}
        ensure // stelle sicher (Vergleich folgt)
          different this[zeile, spalte];
  entail
    // Zur Abwechslung eine Klammerung
    begin
      // Für jede Spalte gilt:
      // Jeder Eintrag ist unterschiedlich.
      forall spalte:{1..::Groesse :Z}
        entail forall zeile:{1..Groesse :Z}
          ensure different this[zeile, spalte];
      // Die Einträge auf der ersten Diagonalen
      // sind alle unterschiedlich
      forall diag:{1..::Groesse :Z}
        ensure different this[diag, diag];
      // Die Einträge auf der zweiten Diagonalen
      // sind alle unterschiedlich
      forall diag:{1..::Groesse :Z}
        ensure different this[diag, Groesse-dia+1];
    end;
  // Für jede Zeile gilt:
  // Die Summe der Einträge ergibt "Summe".
  entail
    forall zeile:{1..::Groesse :Z}
      ensure
        Summe = forall spalte:{1..Groesse :Z}
          apply + to (this[zeile, spalte]);
  // Für jede Spalte gilt:
  // Die Summe der Einträge ergibt "Summe".

```

```

entail
  forall spalte:{1..:Groesse :Z}
    ensure
      Summe = forall zeile:{1..Groesse :Z}
        apply + to (this[zeile, spalte]);
// Für beide Diagonalen gilt jeweils:
// Die Summe der Einträge ergibt "Summe".
ensure
  begin
    Summe = forall diag:{1..:Groesse :Z}
      apply + to (this[diag, diag]);
    Summe = forall diag:{1..:Groesse :Z}
      apply + to (this[diag, Groesse-dia+1]);
  end
end LateinischesQuadrat;

```

2.4 Codegenerierung

Die Problembeschreibung wird in einer Datei gespeichert (`LateinischesQuadrat.csd`) und dann dem ConSolve-Werkzeug `cssh` übergeben. Das Werkzeug `cssh` erzeugt aus dieser Problembeschreibung eine Reihe von C++-Dateien (Codegenerierung). Diese Dateien dienen zur Steuerung des ConSolve-Laufzeitsystems, das später die Lösung des Problems übernimmt. Der Anwender muss die Suche nach Lösungen nur noch anstoßen, weiter hat er damit nichts zu tun.

2.5 Integration in eine Anwendung

Der Code eines aufrufenden Programmes kann sehr einfach sein, wie die Weiterführung unseres Beispiels zeigt:

```

#include "LateinischesQuadrat.hpp"
using namespace ConStruct;
int main ()
{
  int groesse=8; // Groesse des Quadrates parametrisierbar
  int anzahl=3; // Anzahl der gesuchten Lösungen
  LateinischesQuadrat::Groesse groesseP;
  groesseP.defAequivalenz(Predicate(groesse));
  LateinischesQuadrat::LateinischesQuadrat problem;
  LateinischesQuadrat::LateinischesQuadrat::Iterator
    iter1 = problem.iterator();
  int zaehler = 0;
  while (++iter1 && ++zaehler<=anzahl)
  {
    ++zaehler;
    LateinischesQuadrat::LateinischesQuadrat
      loesung = *iter1;
    for(int i=1; i <= groesse; ++i)
    {

```

```

        int summe=0;
        for(int j=1; j <= groesse; ++j)
        {
            int eintrag = loesung.eintrag_i_ii(i, j);
            summe +=eintrag;
            cout.width(3);
            cout.fill('_');
            cout << eintrag << " ";
        }
        cout << "= " << summe << endl;
    }
    cout << endl << endl;
}

```

In diesem Quelltext ist bereits die Ausgabe enthalten. Die Abfrage der Lösung des Beispielproblems geschieht in wenigen Zeilen:

```
#include "LateinischesQuadrat.hpp"
```

bindet die generierte Datei ein.

```
LateinischesQuadrat::Groesse groesseP;
groesseP.setEquivalent(Predicate(groesse));
```

erzeugt ein Prädikat (der universelle Datentyp in ConStruct, dazu später mehr) und weist ihm einen Zahlenwert zu. So kann die gewünschte Größe des Quadrates zur Laufzeit festgelegt werden. Die Deklaration der verwendeten Klassen Groesse, LateinischesQuadrat und LateinischesQuadrat::Iterator (im namespace LateinischesQuadrat) hat bereits in der eingebundenen Datei stattgefunden. Diese Klassen sind aus den exportierten Konzepten in ConStruct entstanden.

```
LateinischesQuadrat::LateinischesQuadrat problem;
LateinischesQuadrat::LateinischesQuadrat::Iterator
    iter1 = problem.iterator();
```

erzeugt ein Lateinisches Quadrat der definierten Größe und einen Iterator über die Lösungen.

```
while (++iter1 && ++zaehler<=anzahl)
```

Mit dieser Schleife werden die Lösungen durchlaufen, wobei nach maximal `anzahl` Lösungen abgebrochen wird. Mit dem Inkrementieren des Iterators wird im ConSolve-System die Suche nach der nächsten Lösung angestoßen. Das Inkrement liefert `false` zurück, wenn keine weitere Lösung gefunden wurde.

```
LateinischesQuadrat::LateinischesQuadrat
    loesung = *iter1;
```

Diese Zuweisung speichert das Ergebnis der Suche.

```
int eintrag = loesung.eintrag_i_ii(i, j);
```

Zur Ausgabe wird das Ergebnis mit dem Selektor `eintrag` abgefragt. Er wurde schon in `ConStruct` angelegt und bei der Codegenerierung in eine C++-Methode übersetzt. Das Suffix `_i_ii` wurde bei der Codegenerierung erzeugt. Es signalisiert den Typ des Rückgabewertes (1 mal Integer) und der Parameter (2 mal Integer). Die Codegenerierung erzeugt auch andere Varianten der Funktion, sie werden hier nicht verwendet.

2.6 Besondere Merkmale

Die wichtigsten Charakteristika von `ConSolve` seien hier nochmals zusammengefasst:

- `ConSolve` wurde speziell für logische und kombinatorische Probleme entwickelt.
- Der Entwickler implementiert keine Lösungsalgorithmen („Deklaratives Prinzip“).
- Die spezielle Problem-Beschreibungssprache `ConStruct` ist nahe an der Ausdrucksweise mathematischer Logik. Dadurch sind Problemformulierungen übersichtlich.
- Es gibt ein effizientes Laufzeitsystem zur Lösungssuche.
- Die constraintbasierte Technik verkleinert die Suchräume.
- Es sind mehrere Strategien für die automatische Lösungssuche konfigurierbar.
- Zwischenstände der Lösungssuche können abgerufen werden.
- Änderungen an der Problemstellung zur Laufzeit werden unterstützt.
- Eine spezielle Shell ermöglicht interaktives Arbeiten ohne Anwendungsprogramm.
- Die Bewertung von Zwischenergebnissen oder extern erstellten Lösungsvorschlägen ist möglich.
- Während der Lösung ist ein Abbruch jederzeit mit einem Zwischenergebnis möglich.
- Die Codegenerierung ermöglicht einfache und schnelle Integration in Anwendungen.
- Die Verwendung einer umhüllten C-Schnittstelle garantiert die Portabilität.

Daraus ergeben sich folgende Vorteile:

- Die Anwendungsentwicklung wird mit `ConSolve` weniger fehleranfällig.
- Der Entwicklungsprozess wird beschleunigt.
- Eine Änderung der Problemstellung zieht nur wenig Wartungsaufwand nach sich.

2.7 Zusammenfassung

Mit ConSolve werden logische und kombinatorische Probleme grundlegend anders angegangen als beim herkömmlichen algorithmischen Programmieren. Der Anwendungsprogrammierer formuliert nur Problembeschreibungen in der domänenspezifischen deklarativen Beschreibungssprache ConStruct und generiert daraus Code für seine Anwendung. Der generierte Code lässt sich in unterschiedliche Programmiersprachen komfortabel einbinden. Die Lösungsalgorithmen sind Bestandteil des ConSolve-Laufzeitsystems, es führt die Suche nach Lösungen selbständig durch und setzt dabei effiziente Verfahren aus der KI ein.

3 Zum Einstieg: Ein genauer Blick auf ein Beispiel

In diesem Kapitel betrachten wir das Beispielpromblem aus Kapitel 2 ausführlich. Die wesentlichen Arbeitsschritte bei der Problemlösung mit ConSolve werden erläutert und die ersten Elemente der Problem-Beschreibungssprache ConStruct an diesem Beispiel vorgestellt.

3.1 „Lateinisches Quadrat“: mathematische/logische Formulierung

Das Lateinische Quadrat aus der Einführung können wir auf mathematische Art beschreiben. Die verwendete Ausdrucksweise wird den meisten Leserinnen und Lesern vertraut sein.

Um später Rechenaufwand zu sparen, versuchen wir zunächst, das Problem zu vereinfachen. Es ergeben sich zwei bedeutsame Vereinfachungen: Die Summe der Zahlen in einer Zeile ist die Summe der Zahlen von 1 bis n , das heißt, sie kann schon vor der Suche berechnet werden. Außerdem ist die Forderung, dass alle Zahlen von 1 bis n genau einmal vorkommen, äquivalent zur Forderung, dass alle Zahlen voneinander verschieden sein sollen und zwischen 1 und n liegen.

Damit ergibt sich folgende Problembeschreibung:

- Die Zahl „Summe“ ist die Summe aller Zahlen von 1 bis n .

$$Summe = \sum_{i=1}^n i$$

- Ein Quadrat besteht aus n mal n Feldern, in jedem Feld steht eine Zahl zwischen 1 und n .

$$\forall i, j (1 \leq i, j \leq n) : 1 \leq Quadrat[i, j] \leq n$$

- Für jede Zeile gilt: Alle Einträge in der Zeile sind paarweise verschieden.

$$\forall i (1 \leq i \leq n) : (\forall 1 \leq j, k \leq n : j \neq k \Rightarrow Quadrat[i, j] \neq Quadrat[i, k])$$

- Für jede Spalte gilt: Alle Einträge in der Spalte sind paarweise verschieden.

$$\forall i (1 \leq i \leq n) : (\forall 1 \leq j, k \leq n : j \neq k \Rightarrow Quadrat[j, i] \neq Quadrat[k, i])$$

- Alle Einträge auf der ersten Diagonalen sind verschieden.

$$\forall i(1 \leq i, j \leq n) : i \neq j \Rightarrow \text{Quadrat}[i, i] \neq \text{Quadrat}[j, j]$$

- Alle Einträge auf der zweiten Diagonalen sind verschieden.

$$\forall i, j(1 \leq i, j \leq n) : i \neq j \Rightarrow \text{Quadrat}[i, \text{Summe} - i + 1] \neq \text{Quadrat}[j, \text{Summe} - j + 1]$$

- Für jede Zeile gilt: Die Summe aller Einträge ist gleich „Summe“.

$$\forall i(1 \leq i \leq n) : \sum_{j=1}^n \text{Quadrat}[i, j] = \text{Summe}$$

- Für jede Spalte gilt: Die Summe aller Einträge ist gleich „Summe“.

$$\forall j(1 \leq j \leq n) : \sum_{i=1}^n \text{Quadrat}[i, j] = \text{Summe}$$

- Für die erste Diagonale gilt: Die Summe aller Einträge ist gleich „Summe“.

$$\sum_{i=1}^n \text{Quadrat}[i, i] = \text{Summe}$$

- Für die zweite Diagonale gilt: Die Summe aller Einträge ist gleich „Summe“.

$$\sum_{i=1}^n \text{Quadrat}[i, \text{Summe} - i + 1] = \text{Summe}$$

Diese Problembeschreibung kann nun fast 1 zu 1 in der ConStruct-Sprache ausgedrückt werden, wie im Beispiel in der Einführung auf Seite 4 zu sehen ist und im folgenden Abschnitt erläutert wird.

3.2 „Lateinisches Quadrat“: Formulierung in der Beschreibungssprache

Unser Beispielproblem beschreiben wir für das ConSolve-System in einer speziellen Sprache¹. Diese Sprache heißt ConStruct und wird hier anhand des oben dargestellten Beispiels eingeführt. Zur Erinnerung: Die komplette Problembeschreibung findet sich in Kapitel 2. Eine vollständige Beschreibung der Sprachelemente und Grammatik findet sich weniger detailliert in Kapitel 8 sowie im „Referenzhandbuch ConSolve/ConStruct“.

```
export Groesse, LateinischesQuadrat > cpp;
```

„export“ bedeutet, dass die bezeichneten Konzepte auch in anderen Modulen sichtbar sind. Ein Modul ist in ConStruct ein Gültigkeitsbereich und Namensraum. In der Regel stimmen Modulgrenzen mit Dateigrenzen überein. In Abschnitt 8.8 wird das Modulkonzept detaillierter erläutert. Der Zusatz

¹„Die Mathematiker sind eine Art Franzosen: redet man zu ihnen, so übersetzen sie es in ihre Sprache, und dann ist es alsobald ganz etwas anderes.“ (Johann Wolfgang von Goethe)

„> cpp“ bewirkt, dass die Konzepte darüber hinaus in eine generierte C++-Header-Datei aufgenommen werden und damit auch von dieser Sprache aus ansprechbar sind. In unserem Beispiel kann die `Groesse` des Quadrates zur Laufzeit festgelegt werden. Außerdem kann auf die Lösungen des „Lateinischen Quadrates“ über einen Iterator zugegriffen werden. Am Ende jedes Sprachkonstruktes steht ein Semikolon.

```
Groesse := 4;
```

Diese Zeile beschreibt ein Prädikat, den universellen Typ für Werte und Bedingungen in `ConStruct`. `Groesse` wird mit dem Wert 4 belegt und implizit als Zahl deklariert. Allgemein wird der Typ jedes Prädikates dynamisch festgelegt.

```
Summe:=forall i:{1..Groesse :Z} apply + to (i);
```

Wir nennen dieses Sprachkonstrukt „Funktionsquantor“. Es steht für die Anwendung einer Operation auf alle Elemente einer Menge. Die `Summe` ergibt sich durch schrittweises Addieren der ganzen Zahlen von 1 bis `Groesse` unter Verwendung der Laufvariablen `i`. Der Funktionsquantor bietet noch mehr Möglichkeiten: Es können statt des `+` auch andere Operatoren verwendet werden (nämlich `*`, `max`, `min`) und statt der Variablen (`i`) können in der Klammer beliebig komplexe Ausdrücken angegeben werden.

Das abgeschlossene Intervall `{1..Groesse :Z}` beinhaltet nur ganze Zahlen. Das `:Z` ist der Standardwert und darf deshalb weggelassen werden. Mit `:R` bezöge sich das Intervall auf reelle Zahlen. Das ist in diesem Fall verboten, denn es wäre eine unendliche Menge aufzuzählen.

```
concept Quadrat
```

```
begin
```

Hier wird ein neues Konzept namens `Quadrat` eingeführt. Konzepte können über ihre Namen angesprochen werden. Sie stehen für eine Menge von Prädikaten. Mit `begin` wird eine Klammerung geöffnet, alle bis zum Schlüsselwort `end` folgenden Prädikate schränken das Konzept ein.

```
entail
```

```
L[{1..:Groesse}, {1..Groesse} : {1..:Groesse}];
```

Die logische Implikation `entail` bedeutet soviel wie „es wird verlangt, dass“. Mit `L` werden Felddefinitionen eingeleitet, mehr dazu in Kapitel 8. `Quadrat` ist damit ein zweidimensionales Feld mit Einträgen zwischen 1 und `Groesse`.

```
export
```

```
eintrag(zeile:{0..}, spalte:{0..}) :T  
:= this[zeile,spalte];
```

Der `eintrag` eines Quadrates ist eine Funktion, die zwei Zahlen als Parameter besitzt und den Wert an der entsprechenden Stelle des Quadrates zurückgibt. Diese Definition ist nur für den Zugriff aus C++ wichtig und wird exportiert, also von außen zugänglich gemacht. `eintrag` steht dann als Methode in C++ zur Verfügung und stellt einen Selektor dar. Der Index hätte statt als Aufzählung auch verschachtelt notiert werden können, beide Möglichkeiten sind äquivalent. Das `:T` steht für den Rückgabotyp, es kann statt der Tautologie („keine Einschränkung“) auch ein Konzeptname angegeben werden. In C++ hat der Rückgabewert dann den Typ dieses (exportierten) Konzeptes. Mit `this` wird das Konzept `Quadrat` selber referenziert, in Anlehnung an C++. Die eckigen Klammern dahinter dienen zur Adressierung der Feldposition.

```
end Quadrat;
```

Am Ende der Konzeptdefinition muss der Name des definierten Konzeptes nochmals genannt werden.

```
concept LateinischesQuadrat
begin
```

```
    inherit Quadrat;
```

Das Konzept `LateinischesQuadrat` erbt vom Konzept `Quadrat` alle seine Einschränkungen und Methoden, z.B. wird der Selektor `eintrag` geerbt.

```
    entail
        forall zeile:{1..:Groesse}
            entail forall spalte:{1..Groesse}
                ensure different this[zeile, spalte];
```

`entail` leitet wieder eine allgemeine Bedingung ein, es folgen zwei verschachtelte Allquantoren. Der Allquantor ähnelt dem oben beschriebenen Funktionsquantor: Beiden gemeinsam ist das Schlüsselwort `forall` und die Angabe einer Wertemenge. Der Allquantor wendet aber keine Funktion an, sondern nimmt eine logische Verknüpfung vor. Die enthaltene Bedingung muss für alle Elemente des Wertebereiches zutreffen.

Im Ausdruck (`this[zeile, spalte]`) wird auf die Laufvariablen beider Quantoren Bezug genommen. Die Verknüpfung der Ausdrücke mittels `ensure different` bedeutet, dass sie alle verschieden voneinander sein müssen. Das Schlüsselwort `ensure` leitet eine Bedingung ein und ist semantisch gleichwertig zu `entail`, syntaktisch gibt es jedoch einen Unterschied: Nach `ensure` folgt ein Vergleich², nach `entail` folgt ein Prädikat.³

Diese Bedingung heißt soviel wie „Für jede Zeile gilt: Alle Einträge in der Zeile sind verschieden voneinander“.

```
    entail
        begin
```

Hier wird eine Gruppe von drei gemeinsam zu erfüllenden Bedingungen mit `begin` und `end` geklammermt.

```
        forall spalte:{1..:Groesse}
            entail forall zeile:{1..Groesse} ensure different this[zeile, spalte];
```

Die erste Bedingung in der Gruppe. ist eine Schachtelung zweier Allquantoren, wie oben (analog für die Spalten).

```
        forall diag:{1..:Groesse}
            ensure different this[diag, diag];
```

Die zweite Bedingung in der Gruppe ist ein einfacher Allquantor. Die Einträge auf der ersten Diagonalen müssen alle unterschiedlich sein.

```
        forall diag:{1..:Groesse}
            ensure different this[diag, Groesse-diag+1];
```

²genauer: eine Infixnotation

³Diese Unterscheidung ist notwendig, um die Grammatik der Sprache `ConStruct` kontextfrei zu halten.

Die dritte Bedingung in der Gruppe ist ebenfalls ein einfacher Allquantor. Wie oben, für die zweite Diagonale.

```
end;
```

Ende der Klammerung der drei Bedingungen.

```
entail
  forall zeile:{1..::Groesse}
    ensure
      Summe = forall spalte:{1..Groesse}
        apply + to (this[zeile, spalte]);
```

Dieses komplexere Prädikat ist ein Allquantor über einen Vergleich zwischen dem einfachen Prädikat `Summe` und einem Funktionsquantor. Wie oben beschrieben: nach `entail` folgt ein Prädikat, nach `ensure` ein Vergleich. Für jede Zeile muss die zuvor berechnete Summe der Zeilensumme entsprechen. (Zur Erinnerung: die Summenbedingungen sind redundant. Sie wurden hier nur aus didaktischen Gründen aufgenommen.)

```
entail
  forall spalte:{1..::Groesse}
    ensure
      Summe = forall zeile:{1..Groesse}
        apply + to (this[zeile, spalte]);
```

Gleiches gilt für die Spaltensummen.

```
ensure
```

Hier wird ein Vergleich eingeleitet. Auch Vergleiche lassen sich mit `begin` und `end` gruppieren.

```
begin
  Summe=forall diag:{1..::Groesse} apply + to (this[diag, diag]);
```

Die Summe der Einträge auf der ersten Diagonalen ist so groß wie `Summe`.

```
Summe=forall diag:{1..::Groesse} apply + to (this[diag, Groesse-dia+1]);
```

Auch die Summe der Einträge auf der zweiten Diagonalen ist so groß wie `Summe`.

```
end
```

```
end LateinischesQuadrat;
```

Ende der Konzeptdefinition.

3.3 Von der Beschreibung zur Lösung: Codegenerierung mit *cssh*

Die Beschreibung unseres ersten kleinen Beispielproblems ist nun fertiggestellt. Im nächsten Schritt generieren wir Hochsprachencode aus der Problembeschreibung und binden den generierten Programmcode in eine Anwendung ein. Derzeit wird nur Code in C oder C++ generiert, die Unterstützung

weiterer Sprachen befindet sich in Vorbereitung. Die Codegenerierung übernimmt das Werkzeug *cssh* (für ConSolve-Shell) bzw. das Werkzeug *cswin*. *cssh* läuft in der Konsole, *cswin* ist ein Programm gleicher Funktionalität mit graphischer Oberfläche. Hier wird nur ein kleiner Teil der Fähigkeiten der Werkzeuge verwendet, mehr zum Funktionsumfang von *cssh/cswin* in Kapitel 7.

Zur Generierung ist lediglich ein parametrisierter Aufruf des Werkzeugs *cssh* erforderlich:
`cssh -q -i LateinischesQuadrat.`

Hier wird vorausgesetzt, dass die Datei `LateinischesQuadrat.csd` im aktuellen Arbeitsverzeichnis liegt und die Problembeschreibung enthält. Als Ergebnis befinden sich danach die Dateien `LateinischesQuadrat.hpp`, `LateinischesQuadrat.c` und `LateinischesQuadrat.h` im Arbeitsverzeichnis. `LateinischesQuadrat.hpp` muss in den Code einer Anwendung eingebunden werden (`include`), die beiden anderen Dateien müssen mit einem C-Compiler übersetzt und der erzeugte Objektcode anschließend zur Anwendung gebunden werden (`linking`). Dem Werkzeug kann mit der Option `-E` ein Verzeichnis mitgeteilt werden, in dem der generierte Code zu speichern ist:

```
cssh -q -E Verzeichnisname -i LateinischesQuadrat.
```

Die Verwendung des generierten Codes in einem Anwendungsprogramm wurde bereits in Kapitel 2 beschrieben. Hier verwenden wir nur einen kleinen Teil der Schnittstelle, zu weiteren Möglichkeiten der Kommunikation einer Anwendung mit dem ConSolve-System siehe Kapitel 4.

4 Zum Kennenlernen: Integration und Kommunikation

In diesem Kapitel geht es um die Integration des ConSolve-Systems als Komponente in ein Anwendungsprogramm und um die Möglichkeiten, das ConSolve-System aus einem Anwendungsprogramm heraus zu steuern.

4.1 Codegenerierung mit *cssh* oder *cswin*

Das ConSolve-Laufzeitsystem haben wir aus Gründen der Portabilität mit einer C-Schnittstelle versehen. Eine Codegenerierung ermöglicht die transparente Verwendung von ConStruct-Konzepten in einer Hochsprache. Am Beispiel C++ sieht das so aus: Sämtliche Elemente der Problembeschreibung, die eine Anwendung verwenden soll, existieren als Klassen und Methoden in dieser Sprache. Die notwendigen Aufrufe der C-Schnittstelle führt generierter Code aus, vor dem Anwender bleibt diese Schnittstelle verborgen.

Die Werkzeuge *cssh* und *cswin* generieren aus der Problembeschreibung den nötigen Code für das Anwendungsprogramm. Beide Werkzeuge verfügen über identische Möglichkeiten. Das Programm *cswin* besitzt eine graphische Benutzeroberfläche, wohingegen *cssh* eine Konsolenanwendung ist. Zur reinen Codegenerierung empfehlen wir, *cssh* zu verwenden.

Beide Werkzeuge ermöglichen auch die interaktive Arbeit mit dem ConSolve-Laufzeitsystem, der ein eigenes Kapitel gewidmet ist. Hier wird nur geschildert, wie man mit *cssh* die notwendige Codegenerierung durchführt und wie man die generierten Dateien verwendet, um ConSolve in ein Anwendungsprogramm zu integrieren.

Eine ausführlichere Darstellung dieses Themas findet sich im Referenzdokument „Schnittstelle von ConStruct zu den Programmiersprachen C und C++“ und in Abschnitt 9.1 „Warum die C-Schnittstelle“.

4.1.1 Vorbedingungen

- Unter Windows muss sich *cssh.exe* im aktuellen Pfad befinden oder über die Pfadvariable zu finden sein. Unter Linux gilt dasselbe für *cssh*.
- Zum Übersetzen müssen die mitgelieferten Dateien *cscpp.hpp* und *csc.h* in den Include-Verzeichnissen enthalten sein, denn der generierte Code bindet sie ein.
- Zur Laufzeit muss die dynamische Bibliothek *consolve.dll* im Programmverzeichnis vorhanden sein.

- Wir gehen davon aus, dass sich der ConStruct-Code zur Problembeschreibung in der Datei `Problem1.csd` im Arbeitsverzeichnis befindet.

4.1.2 Aufruf

In unserem Beispiel stoßen wir die Codegenerierung durch *cssh* mit dem Befehl `cssh -q -i Problem1` an.

Der Parameter `-q` bedeutet, dass sich die Shell nach der Generierung selbst beendet. Der Parameter `-i Problem1` bedeutet, dass ein Construct-Modul `Problem1` importiert werden soll. Beim Import findet auch die Codegenerierung statt. Der Modulname wird automatisch um die Endung `„.csd“` zum Dateinamen ergänzt.

Die Codegenerierung umfasst bei diesem Aufruf genau die in der Problembeschreibung definierten Exporte für den Zugriff von außen.

Das Kapitel 7 beschreibt alle erlaubten Aufrufparameter und weitere Möglichkeiten.

4.1.3 Ergebnis

Das Werkzeug *cssh* generiert drei Dateien, die sich alle im aktuellen Verzeichnis befinden:

Problem1.hpp ist eine C++-Datei, die alle in der Problembeschreibung als `export > cpp` gekennzeichneten Konzepte als Klassen deklariert und definiert. Sie reicht jeden Zugriff auf eine ihrer Methoden „inline“ als Aufruf einer C-Funktion weiter.

Problem1.c ist eine C-Datei. Sie definiert die C-Funktionen, die `Problem1.hpp` aufruft, und spricht die ConSolve-Laufzeitbibliothek über ihre C-Schnittstelle an.

Problem1.h ist eine C-Headerdatei. Sie enthält die für `Problem1.c` notwendigen Deklarationen.

Der Inhalt der beiden letztgenannten Dateien ist für den Anwendungsentwickler in aller Regel nicht von Interesse.

4.1.4 Verwendung der Dateien

Die Datei `Problem1.hpp` wird vom Entwickler in seine C++-Anwendung eingebunden und erlaubt die transparente Verwendung der generierten Klassen und Methoden.

Die Datei `Problem1.c` muss übersetzt werden, sie bindet dabei `Problem1.h` ein. Die entstehende Objektdatei `Problem1.o` muss zur Anwendung gebunden werden. Bei Verwendung einer IDE gilt: Die Dateien `Problem1.c` und `Problem1.h` müssen Bestandteil des Projektes sein.

4.2 Die Verwendung exportierter Konzepte unter C++

4.2.1 Der Iterator

Der Zugriff auf Lösungen ist in ConSolve mit Iteratoren realisiert. Ein Iterator über ein Konzept hat den Typ eines Zeigers auf eine Lösung dieses Konzeptes. Zu jedem exportierten Konzept wird der Iteratortyp automatisch deklariert. Mit dem Anlegen eines Iterators wird noch keine Suche nach einer Lösung angestoßen. Ein Iterator lässt sich inkrementieren und zeigt danach auf die jeweils nächste Lösung, bei Optimierung auf die nächste bessere Lösung. Die Anzahl der Elemente, über die ein Iterator läuft, ist in aller Regel nicht bekannt. Gibt es keine weitere (bessere) Lösung, so stellt der Iterator nach der Suche keinen gültigen Zeiger dar. Dieser Status kann abgefragt werden. Dereferenziert man den Iterator trotzdem, so wird eine Ausnahme geworfen. Die Fehlerbehandlung wird in Abschnitt 4.7 beschrieben.

Zur Erläuterung ein vereinfachter Auszug aus unserem Beispiel:

```
LateinischesQuadrat::LateinischesQuadrat problem;
LateinischesQuadrat::LateinischesQuadrat::Iterator
    iter1 = problem.iterator();
// Abfrage aller Lösungen,
// also Abbruch, wenn Nullzeiger:
while (++iter1)
{
    LateinischesQuadrat::LateinischesQuadrat
        loesung = *iter1;
    /* Ausgabe der Lösung ... */
}
```

Wenn das Problem überbestimmt ist und keine Lösung besitzt, dann gibt schon das erste Iterator-Inkrement den Nullzeiger zurück.

Bei der Lösungssuche werden alle Bedingungen verwendet, die Bestandteil der Problembeschreibung sind. Weitere Bedingungen können über Konstruktoren erzeugt und einem Iterator übergeben werden, siehe Abschnitt 4.3. Nur dieser Iterator berücksichtigt die zusätzlichen Bedingungen bei der Lösungssuche, andere Iteratoren über dasselbe Konzept müssen die zusätzlichen Bedingungen gegebenenfalls auch erhalten. Hat ein Iterator zusätzliche Bedingungen erhalten, so können sie auch wieder entfernt werden; das ist ein Unterschied zu den Bedingungen aus der Problembeschreibung.

Die Einstellungen zur Lösungssuche werden einem Iterator einmalig bei seiner Erzeugung mitgegeben, vergleiche dazu Abschnitt 6.2. Zur Laufzeit ist keine Änderung der Suchparameter mehr möglich, es müssen gegebenenfalls neue Iteratoren angelegt werden. Es können für eine Problembeschreibung bzw. ein Konzept mehrere Iteratoren existieren, die unterschiedliche Suchparameter verwenden.

4.2.2 Der Export von Prädikaten, Konzepten und Methoden

Problembeschreibungen in der Sprache ConStruct können Elemente exportieren. Der Export eines Elementes ermöglicht die Verwendung dieses Elementes auch außerhalb der Problembeschreibung selbst.

ConStruct kann zwei Arten von Elementen exportieren:

- Ganze Konzepte können exportiert werden.
Diese Konzepte werden mit all ihren Prädikaten und Methoden exportiert.
- Einzelne Prädikate, Selektoren oder Konstruktoren können exportiert werden.

Exporte haben drei Bedeutungen:

- Bei der Codegenerierung wird der Zugriff auf das exportierte Element ermöglicht, wenn eine Zielsprache angegeben wurde.
Als Zielsprachen sind derzeit C, C++ und HTML möglich. Der Export nach HTML erzeugt eine Beschreibung des exportierten Konzeptes und ist zu Dokumentationszwecken nützlich. Der Export nach C++ impliziert aus technischen Gründen den Export nach C.

Beispiel:

```
export Konzept1 > cpp, html;
concept Konzept1
begin
// Konzeptdefinition, exportiert nach C, C++ und HTML
end Konzept1;
```

- Exportierte Elemente können in der interaktiven Arbeit mit dem Werkzeug *cssh/cswin* direkt angesprochen werden. Dafür ist die Angabe einer Zielsprache nicht erforderlich.

Beispiel:

```
export Konzept1
// weiter wie oben
```

Eine Eingabe am Prompt könnte so aussehen:

```
cssh> import*Problem1 (Modul Problem1 importieren)
cssh> ^Konzept1      (Iterator über Konzept1 anlegen)
cssh> +              (letzte Variable (hier: Iterator) inkrementieren)
```

Die interaktive Nutzung wird in Kapitel 7 beschrieben.

- Exportierte Elemente sind in anderen ConStruct-Modulen sichtbar.
Wir betrachten zunächst nur Problembeschreibungen aus einem einzigen Modul. Das Modulkonzept in ConStruct wird in Abschnitt 8.8 näher erläutert.

4.2.3 Namenskonventionen

Alle Namen sind mit dem ConStruct-Modulnamen qualifiziert.

Exportierte Konzepte und einzelne Prädikate finden sich unter C++ als Klassen des entsprechenden Namens wieder. Exportierte ConStruct-Methoden und Prädikate innerhalb eines Konzeptes werden in C++ zu Methoden und Datenelementen der Konzeptklasse.

Alle Funktionsnamen haben als Suffix den Typ ihres Rückgabewertes und ihrer Parameter, jeweils hinter einem Unterstrich. Dabei steht der Buchstabe „i“ für Integer, „d“ für Double, „s“ für String und „p“ für Prädikat.

Beispiel:

Der ConStruct-Selektor `eintrag` in

```

concept Quadrat
begin
  entail L[{1..4 :Z},{1..4 :Z}:{1..16 :R}];
  eintrag(zeile:{0.. :Z}, spalte:{0.. :Z}) :T
    := this[zeile,spalte];
end Quadrat;

```

wird unter C++ zur Methode

```

Modulname::Quadrat::eintrag_d_ii(zeile, spalte)

```

Gegebenenfalls werden auch andere Namensvarianten der Funktion erzeugt, mehr dazu findet sich im Abschnitt 9.1 „Warum die C-Schnittstelle“.

4.3 Zusätzliche Bedingungen einfügen

Wie schon angedeutet, kann eine Anwendung die Problemstellung zur Laufzeit ergänzen. Dazu wird ein Prädikat erzeugt und seine Referenz einem Iterator übergeben. Diese Zuweisung einer Bedingung geschieht mit einem Operator (in C++):

```

Predicate::Iterator&
  Predicate::Iterator::operator &= (const Predicate&)

```

Das funktioniert für Referenzen auf beliebige Prädikate. Die Konstruktion eines Prädikates ist allerdings eingeschränkt: Über die Funktionen der Schnittstelle können Prädikate nur aus Einzelwerten von Aufzählungstypen erzeugt werden.

In C++ stehen dafür folgende Funktionen zur Verfügung:

- `ConStruct::Predicate (int i)`
erzeugt ein Prädikat aus einer ganzen Zahl.
- `ConStruct::Predicate (double d)`
erzeugt ein Prädikat aus einer Fließkommazahl.
- `ConStruct::Predicate (const std::string &sym)`
erzeugt ein Prädikat aus einer Zeichenkette.
- `ConStruct::Predicate (const Predicate &p)`
erzeugt ein Prädikat als Kopie eines Prädikates.

Komplexe Prädikate müssen in `ConStruct` durch einen exportierten Konstruktor beschrieben werden, wenn Instanzen davon in C++ erzeugt werden sollen.¹ Mehr dazu in Abschnitt 5.6.

Man kann ein nachträglich eingefügtes Prädikat auch wieder löschen. Dazu ruft man eine Methode auf:

¹Der Grund dafür liegt in der Prüfung auf Zulässigkeit, die das Werkzeug `cssh` bei der Codegenerierung vornehmen kann.

```
int Iterator& remove(const Predicate& constraint);
```

Wichtig ist dabei, dass das angegebene Prädikat genauso angelegt wurde (genauer: dieselbe Normalform hat) wie das zuvor eingefügte². Wird das zu löschende Prädikat identifiziert und entfernt, so ist der Rückgabewert der Methode von Null verschieden. Bei Misserfolg ist der Rückgabewert Null.

Ein neu angelegter Iterator hat natürlich keines der Prädikate, die anderen Iteratoren zuvor übergeben wurden.

4.4 Manuell entstandene Lösungsvorschläge

Oftmals soll die Interaktion eines Anwendungsprogrammes so weit gehen, dass der Benutzer eine maschinell generierte Lösung des Problems nachträglich verändern oder sogar eine initiale Lösung zur maschinellen Verbesserung vorgeben kann. ConSolve kann eine manuell entstandene Lösung entsprechend der Problembeschreibung bewerten, kann die verletzten Bedingungen auflisten und kann von der manuellen Lösung ausgehend bessere Lösungen suchen.

4.5 Beobachter im ConSolve-System

Im ConSolve-System gibt es eine Beobachter-Klasse, die der Anwendungsentwickler verwenden kann.³ Diese Klasse trägt den Namen `ConStruct::Predicate::Iterator::Listener` und sorgt für einen Informationsfluss zwischen Laufzeitsystem und Anwendung. Ein solcher Beobachter wird von der Anwendung erzeugt und einem Iterator übergeben. Ein Iterator darf mehrere Beobachter erhalten.

Der Beobachter besitzt Methoden, die in bestimmten Stadien der Suche aufgerufen werden. Der Rückgabewert dieser Methoden beeinflusst den weiteren Verlauf der Suche. Spezialisierte Beobachterklassen können diese Methoden redefinieren und die Suche nach den Wünschen des Anwenders beeinflussen.

Der Beobachter wird als `const`-Referenz übergeben und der Iterator legt von jedem Beobachter eine Kopie an, über deren Existenz er selber bestimmt. Der Beobachter kann nur noch über Zeiger mit dem Rest der Anwendung außerhalb der Laufzeitbibliothek kommunizieren. Das Werkzeug *cswin* zeigt diese Praxis: Der Beobachter benachrichtigt ein Fensterobjekt, das den Fortgang der Suche anzeigt. Die Nachricht zum Abbruch der Suche wird vom Fenster an den Beobachter übermittelt.

Die Standardimplementation des Beobachters gibt lediglich eine Fortschrittsinformation auf der Fehlerkonsole aus. Die Einflussmöglichkeiten spezialisierter Beobachter reichen bis zur Auswahl des nächsten Suchschrittes durch externe Funktionen.

²Wiederholtes Einfügen desselben Prädikates wird erkannt und nicht ausgeführt. Eine Ausnahme sind Prädikate mit nicht idempotenten Maßen.

³Exakt ausgedrückt: Es gibt mehrere Beobachterklassen, von denen eine öffentlich ist.

4.5.1 Die Funktionen der Beobachter

Im Folgenden betrachten wir die öffentlichen Methoden der Beobachter, die während der Suche aufgerufen werden. Spezialisierte Beobachterklassen müssen diese Methoden redefinieren. Gemeinsam ist diesen Methoden, dass sie einen Integer-Wert zurückgeben: Die Rückgabe einer Eins bedeutet, dass die Suche fortgesetzt werden soll und die Rückgabe einer Null bedeutet, dass die Suche geordnet abgebrochen werden soll.

Es spricht nichts dagegen, an dieser Stelle komplexe Abbruchkriterien auszuwerten oder mit dem Anwender zu interagieren. Bei Zeitmessungen zum Testen der Performanz werden die Ausführungszeiten der Beobachter allerdings mitgezählt. Die Beobachter müssen aber auf jeden Fall zurückkehren, damit die ConSolve-Bibliothek ihren Lauf ordnungsgemäß beenden kann und keine allozierten Objekte im Speicher verbleiben.

onNewDepth

Diese Methode ruft der Iterator auf, wenn im Suchbaum eine neue Tiefe erreicht wird. Die Rückgabe einer Null bedeutet den Abbruch der Suche, ansonsten wird sie fortgesetzt.

Signatur:

```
virtual
int ConStruct::Predicate::Iterator::Listener
    ::onNewDepth( int numVerzw,
                 int tiefe,
                 int varsAufAgenda,
                 const Valuation& abstand,
                 const Valuation& akzeptanz )
```

Parameter:

numVerzw ist die Anzahl der bisher durchgeführten Verzweigungen.

tiefe ist die momentane Tiefe im Suchbaum.

varsAufAgenda ist die Anzahl der Variablen, über deren Wertebereich noch mindestens einmal verzweigt werden muss. $tiefe/(tiefe+varsAufAgenda)$ ist daher eine Schätzung für den Anteil der bereits erledigten Sucharbeit.

abstand ist eine untere Schranke für die Bewertung im gerade durchsuchten Zweig. Wenn hier eine Lösung gefunden wird, kann sie nicht besser bewertet sein.

akzeptanz ist die obere Schranke für die Bewertung. Ein Ergebnis mit dieser Bewertung wurde schon gefunden und schlechtere Ergebnisse werden nicht akzeptiert.

onOptStep

Diese Methode wird aufgerufen, wenn ein lokales Suchverfahren einen Verbesserungsschritt einleitet.

Signatur:

```
virtual
int ConStruct::Predicate::Iterator::Listener
    ::onOptStep( int step,
                const Predicate& alteBelegung )
```

Parameter:

step ist eine fortlaufende Nummer der Verbesserungsversuche.

alteBelegung beschreibt die Teile der alten Lösung, die in der Optimierungsregion liegen.

onSolution

Diese Methode wird aufgerufen, wenn eine neue Lösung oder Verbesserung gefunden wurde.

Signatur:

```
virtual
int ConStruct::Predicate::Iterator::Listener
    ::onSolution( const Valuation& valuation,
                 const Predicate& solution )
```

Parameter:

valuation gibt die Bewertung der Lösung an.

solution beschreibt die gefundene Lösung als dichotomisches⁴ Prädikat.

4.6 Suchbaum schreiben

Das ConSolve-Laufzeitsystem kann die Baumsuche protokollieren. Dabei wird bei jeder Verzweigung während der Baumsuche eine entsprechende Information in eine Suchbaumdatei geschrieben. Das Analyse-Werkzeug CSV („ConSolveViewer“) kann solche Suchbäume anzeigen. Die Analyse kann darüber Aufschluss geben, wie effizient eine Suche abgelaufen ist und woraus die einzelnen Schritte der Suche bestanden haben.

Der Suchbaum wird durch Aufruf der folgenden Funktion erzeugt:

fileOutSearchtree

Signatur:

```
void Iterator::fileOutSearchtree(const std::string& filename,
                                int withPropInfo=0)
```

Parameter:

⁴Dieser Fachbegriff wird im Glossar erläutert.

filename gibt den Dateinamen an. Die erzeugte Datei besitzt die Endung „.tre“. Ohne Pfadangabe wird die Datei im Arbeitsverzeichnis angelegt. Eine bereits existierende Datei wird überschrieben.

withPropInfo gibt an, ob zu jeder Verzweigung eine Information über die durchgeführte Propagierung abgelegt werden soll. Das kann in Problemfällen sehr hilfreich sein. Die erzeugten Daten sind aber oft sehr umfangreich! Zum Protokollieren der Propagierungsinformation wird ein Wert ungleich Null übergeben.

4.7 Fehlerbehandlung

Bei Auftreten eines Fehlers wirft das ConSolve-System eine Ausnahme. Die Ausnahmen sind im Referenzdokument „Schnittstelle von ConStruct zu den Programmiersprachen C und C++“ eingehender beschrieben. Hier beschränken wir uns auf die Angabe der Typen der Ausnahmen, der Fehlernummern und einer kurzen Beschreibung.

4.7.1 Die Typen der Ausnahmen

Die Ausnahmen werden von der C++-Hülle um die Schnittstelle geworfen, wenn eine der aufgerufenen C-Funktionen einen entsprechenden Wert zurückgibt. Bei Verwendung der C-Schnittstelle werden natürlich keine Ausnahmen geworfen. Die interne Fehlerbehandlung des ConSolve-Systems wird hier nicht besprochen.

Es gibt vier Typen von Ausnahmen:

ConStruct::Object::Error ist die allgemeine Fehlerklasse. Die drei anderen Fehlerklassen sind Spezialisierungen dieser Klasse.

ConStruct::Valuation::Error wird geworfen, wenn bei der Verarbeitung einer Bewertung ein Fehler auftritt.

ConStruct::Predicate::Error wird geworfen, wenn bei der Verarbeitung eines Prädikates ein Fehler auftritt.

ConStruct::Predicate::Iterator::Error wird geworfen, wenn bei der Handhabung eines Iterators ein Fehler auftritt.

Die Fehlerklassen besitzen folgende Methoden:

msg

Diese Funktion gibt einen Wert aus einem Aufzählungstyp zurück. Der Rückgabewert entspricht einer der unten aufgelisteten Fehlernummern.

Signatur:

```
ConStruct::Object::Error::E_Nr ConStruct::Object::Error::msg( ) const
```

details

Diese Funktion gibt einen beschreibenden Text zurück.

Signatur:

```
std::string ConStruct::Object::Error::details( ) const
```

object

Die drei speziellen Fehlerklassen besitzen zusätzlich eine Methode `object()`. Sie gibt einen Verweis auf das betreffende Objekt zurück. Zum Beispiel liefert `ConStruct::Predicate::Error` eine Referenz auf das Prädikat, bei dessen Verarbeitung der Fehler aufgetreten ist.

4.7.2 Fehlernummern und Kurzbeschreibung

1 GeneralError

Ein nicht näher spezifizierter Fehler ist aufgetreten.

2 NotYetImplemented

Die Funktionalität ist noch nicht implementiert.

3 ValuationNotANumber

Eine komplexe Bewertung sollte in eine Zahl umgewandelt werden.

4 UnknownHierarchyLevel

Eine angefragte Hierarchieebene der Bewertung ist nicht vorhanden.

5 ExpectValueHere

Beim Aufruf einer Methode wurden zuwenig Parameter übergeben.

6 TypeError

Die Umwandlung eines komplexen Prädikates in einen Basistyp ist nicht erlaubt.

7 SolutionNotAvailable

Ein Iterator wurde dereferenziert, obwohl er nicht auf eine Lösung zeigt.

8 IllegalAttributeName

Ein unbekanntes Attribut eines Prädikates wurde angefordert.

9 UnknownModuleVersion

Es liegt ein Versionskonflikt zwischen Codegenerierung und Lösungssuche vor.

10 ErrorInstallingModule

Ein Modul konnte nicht geladen werden.

11 FileError

Eine Datei konnte nicht geöffnet werden.

4.7.3 Fehlerbehandlung in der C-Schnittstelle

Funktionsaufrufe in der C-Schnittstelle zeigen das Auftreten eines Fehlers über ihren Rückgabewert an. Nach dem Auftreten eines Fehlers kann die Nummer des zuletzt aufgetretenen Fehlers und ein beschreibender Text über die Schnittstelle abgefragt werden. Die Funktionen und ihre Rückgabewerte sind im Referenzdokument „Schnittstelle von ConStruct zu den Programmiersprachen C und C++“ beschrieben.

4.7.4 *Callback*-Mechanismus als Alternative

Über die C-Schnittstelle kann ein Zeiger auf eine Funktion übergeben werden, die bei Auftreten eines Fehlers aufgerufen werden soll. Diese Funktion muss den Kontrollfluss wieder an das Laufzeitsystem zurückgeben, sonst bleiben Speicherbereiche alloziert. Es wird nur ein einziger Funktionszeiger gespeichert. Der *Callback* wird durch Übergabe eines Nullzeigers gelöscht.

Über die C++-Schnittstelle ist die Angabe einer *Callback*-Funktion ebenfalls möglich. Wird ein solcher *Callback* gesetzt, so ist werden keine Ausnahmen mehr geworfen. Die Übergabe eines Nullzeigers als *Callback* aktiviert die Ausnahmen wieder.